

## Assignment 4

**Submission.** Your GitHub classroom submission will look *something like* the following structure:

```
matcher/
  data/
    queries.tsv
    annotated-index.tsv
    annotated-index-train.tsv
    annotated-index-test.tsv
  matcher/
    __init__.py
  bin/
    __init__.py
    main.py
  name_matcher.py
  utils/
    __init__.py
    parse_tsv.py
  eval/
    __init__.py
    eval.py

tests/
  data/
    test_names.tsv
    __init__.py
    test_matcher.py

.gitignore
README.md
setup.py
pytest.ini
discussion.md
logs.txt
environment.yml
```

**Problem 1.** In this problem, you will write code for the task of Fuzzy Name Matching. Imagine that your first project in your new role as a natural language processing engineer is to develop an algorithm that, given two name strings, produces a score ranging from 0 to 1 describing their similarity. For example, the strings “Joe” and “Joe” should receive a score of 1 since they are an exact match, while “Joe” and “Mary” should receive a score of 0 since they have no matching characteristics (e.g., their characters). Consider the following questions. What happens when two strings are similar, but not identical? How do we reason about assigning similarity scores when these very common situations arise? Throughout this problem, we will investigate this challenge.

There are a variety of situations where this task might arise. For example, a database may contain duplicate entries for the same entity due to a misspelling in the name. Another application is record linkage, where you wish to merge multiple records together based on, say, the name field. **All discussion questions should be answered in `discussion.md`. This is a large dataset, so you can sample 10% of the data from the training set and 10% of the data from the test set and tackle the reduced problem. Just make sure to note this in the discussion file. You should fix a random seed to make this process repeatable.**

(A) Your first question for any artificial intelligence problem should be “what does the data look like?” You ask your team, and the dataset is still being put together by another team responsible for acquiring and processing the data. How frustrating. But you still need to make headway on this project, so you think about prototyping the real data with a similar dataset. Lucky for you, your team has some code lying around to create a prototype dataset. Often, it’s important to demonstrate your method works on prototype data. Your teammate already used the Wikidata query language SPARQL to generate one table of names of U.S. politicians and a second containing politician names with some alias information pulled from Wikidata. There’s no need to understand the SPARQL code used to generate these tables; simply navigate to <https://query.wikidata.org/> and insert the block below to get a sense of how the dataset was created. You can click the blue arrow on the left-hand side to execute the query. Then, you should obtain a table of 250 politician names in English. For this assignment, you won’t use the table you’ve generated; instead, you’ll use the two tables, `queries.tsv` and `annotated-index.tsv`, uploaded to Canvas. **Your goal will be to perform matching over the query list names.** The annotated index was built as follows: for each of the 250 names in the query list, for each of the 10,000 index list names, if there is a match, record a “1” in the row; otherwise, record a “0” in the row. The SPARQL query to generate the query names is

```
SELECT DISTINCT ?personLabel
WHERE {
  ?person wdt:P106 wd:Q82955;
          rdfs:label ?personLabel.
  FILTER(LANG(?personLabel) = "en") .
}
LIMIT 250
```

(B) Now that you have a dataset, it's time to explore a few different approaches. We always start with a simple approach, which we think of as our baseline. What's the simplest name matching algorithm you can devise? The simplest technique is exact match, where a score of 1 is returned if the two inputs are exactly the same, and 0 is returned otherwise. Create a Python project following the structure above. Inside of `name_matcher.py`, create a `NameMatchScorer` interface like the one below:

```
class NameMatchScorer:
    """Interface for scoring putative name matches"""
    def __init__(self, name1, name2):
        self.name1 = name1
        self.name2 = name2

    def __str__(self):
        # TODO - fill in (see our work from corpora.py for an example)

    def __repr__(self):
        # TODO - fill in (see our work from corpora.py for an example)

    def score(self, name1, name2):
        pass
```

Next, create a subclass of `NameMatchScorer` that performs exact match scoring. This means the subclass will inherit from the base class and overwrite the `score` method. The names in your data table can be read off directly from the file, so there's no need to stick them inside of some other data structure. To evaluate how well your exact match does, write a function in `eval.py` that produces precision, recall, and F-1 scores. Note that since we know when `personLabel` and `aliasLabel` refer to the same person, we trivially obtain perfect precision classifiers because there are no false positives. You should use an off-the-shelf library to do all this. You might enclose this logic in a function called `evaluate` in `eval.py`. The function `evaluate` should take two arguments: the first is the results you wish to score, and the second is the string name for a metric (namely, precision, recall, or F-1).

We need to take a moment to think about how we are going to iterate over this dataset. As we said previously, `pandas` does not scale well to larger datasets. Our `queries.tsv` file is certainly not a large dataset, but perhaps we should be thinking ahead: it might be the case that the data team gives us an enormous file, so we don't want to tie ourselves to a library that we know doesn't scale.

One thing we might choose to do is simply iterate over every row in the table, computing a similarity score at each position. An approach to doing this—without storing information in shaky data structures like the `pandas DataFrame`—is to define a custom iterator. Here's the basic idea: we want to proceed through the file and perform scoring at each row in memory. We will define an iterator to parse the tsv file for us. As we alluded to in class, iterators expect to have certain special methods that define behaviors we'd want to be true of an iterator. For instance, `__next__` is a special method that we'll need to define in our custom iterator to describe how to proceed across the tsv file we're traversing. We'll also create a custom object to store the information in each row, along with the corresponding score. All this has been done for you below. Copy the following into `parse_tsv.py`. If you are using a format other than tsv, you will have to make some minor adjustments.

```
import abc
import csv
import collections.abc
import dataclasses

from matcher.name_matcher import NameMatchScorer

@dataclasses.dataclass
class Comparison:
    """Class representing two names and their similarity"""
    name1: str
    name2: str
    score: float = 0.0

class DocIterator(abc.ABC, collections.abc.Iterator):
    def __str__(self):
        return self.__class__.__name__

class TsvIterator(DocIterator):
    """Iterator to iterate over tsv-formatted documents"""
    def __init__(self, path):
        self.path = path
```

```
self.fp = open(self.path)
self.reader = csv.reader(self.fp, delimiter='\t')
next(self.reader) # skip first row

def __iter__(self):
    return self

def __next__(self):
    try:
        row = next(self.reader)
        return Comparison(row[0], row[1], NameMatchScorer(row[0], row[1]))
    except StopIteration:
        self.fp.close()
        raise
```

The main addition you will have to make here is writing the `Comparison` objects to a new file. From that file, you will be able to perform scoring. Separately, you could make changes to this file by doing all the scoring in memory and not writing to a file, but it might be easier to have a file of results to score from.

Now, we'll use a dataset split to find good thresholds. Make sure you split the `annotated-index.tsv` file into an 80-20 training-test split. Note that you shouldn't do this based on lines (i.e., the first 40,000 lines are training and the remaining are used for testing). Since multiple rows can correspond to a single entity, you need to find the number of unique entities and split from there. For instance, if there are 1,300 unique names, then the first 80% can be used for training, with the remaining 20% forming the test dataset. What are your precision, recall, and F-1? Call your evaluation for this approach (and the following ones) from the `main.py` module, where you'll store the code that actually puts together and runs everything, as we've seen before. So, you will likely have an import similar to `from matcher.eval import evaluate`. Write your findings in `discussion.md` at the top-level of your package. When you write your precision, recall, and F-1, place them inside of a table in the markdown. You'll be scoring a few other methods, so it will be helpful to present your work in an organized way, such as:

	Precision	Recall	F-1
Exact Match			
Jaccard Similarity			
Levenshtein			
Cosine Similarity			

(C) Let's make things a little more interesting by writing a method that computes the Jaccard similarity between two name strings. Given strings  $s = s_1s_2...s_n$  and  $t = t_1t_2...t_m$ , the Jaccard similarity between  $s$  and  $t$  is computed as  $score = |\text{intersection}(s,t)| / |\text{union}(s,t)|$ . So, the numerator is the number of elements shared between  $s$  and  $t$ , while the denominator is the total number of unique elements across both strings. As in (B), your Jaccard similarity class should inherit from `NameMatchScorer`, and you should evaluate in `main.py` using the evaluation code you write in `eval.py`. But now we have to make a decision we didn't have to worry about in (B). We have to decide what level of similarity constitutes a match. The threshold value should be an optional parameter for scoring, so this is what your `__init__` method on `NameMatchScorer` should look like now:

```
class NameMatchScorer:
    """Interface for scoring putative name matches"""
    def __init__(self, name1, name2, threshold=0.5):
        self.name1 = name1
        self.name2 = name2
        self.threshold = threshold
```

Then, your Jaccard similarity scorer (and some others) will use the threshold value after inheriting from `NameMatchScorer`. Notice that with the exact match method above, intuitively thresholding doesn't have any effect. Whether you increase or decrease the threshold, the precision will be maximized and the recall will be minimized.

- (D) As we said above, the threshold is set as a default of 0.5. What would we expect to happen if we changed the threshold? For instance, what does a threshold of 1.0 mean intuitively? What about a threshold of 0.01? Put your responses in `discussion.md`, and make sure you reference precision and recall.
- (E) Now, how should we find the optimal threshold? In practice, this threshold is best determined by end-users of our application. Some may prefer higher precision; others may prefer higher recall. Still, we can make an initial thresholding decision on the basis of precision and recall. We can pick a threshold of 0.5 for some scoring schemes, but 0.5 doesn't always jibe with the particular similarity scoring. In particular, plot precision and recall as a function of the threshold using `precision_recall_curve` from the `sklearn.metrics` library. Your plot should vary the threshold along the x-axis, with precision and recall as curves on the plot, so the y-axis will range from 0 to 1. What is the optimal threshold for the Jaccard similarity? Once you have your optimal threshold, comment on the precision, recall, and F-1.

- (F) So far, we’ve gotten our hands on data that represents our project goals and we have a few simple baselines, like exact match and Jaccard similarity. Our next approach will use the Levenshtein Distance. This is a metric that records the minimum number of transformations (insertions, deletions, or substitutions) needed to map one name to another. You should use an off-the-shelf approach here, such as the `python-Levenshtein` library available [here](#). Does it make sense to use the same threshold as you did for the Jaccard similarity here? It does not, so you’ll want to be sure to adjust that setting in the method for the Levenshtein distance by applying the same work you completed in Part (E). Write that threshold in `discussion.md`. Then, report the precision, recall, and F-1 in your table. As you’re putting together your `name_matcher` package, you’re thinking about the project organization, but you should also consider how a user interacts with your work. For example, it might be convenient to support the following application programming interface (API), which might make it simple to run experiments (e.g., “how does exact match compare to Jaccard similarity?”).

```
python main.py -f <path_to_dataset> -s <scoring_algorithm> -e
<flag_whether_to_evaluate> -p <flag_whether_to_print_results>
```

How you design the API (for example, you’ll probably use `argparse` and command-line options like the ones above) is up to you, and the simple interface here is just one example. You might come up with something different, but the take-home message is that you should always keep the user interface in mind. So, for example, you aren’t required to add in code to print out your results with `-p` as above, but it might be really helpful for you or a user to see results of the form `(s, t, score, matcher)`, where `s` and `t` are the names being compared and `score` is the score from running `matcher`.

- (G) At this point, you’ve worked with the dataset a bit and run a number of different baselines. You’ve also labored to set your work up in a well-formatted Python project. Good job. You now have a sense of the difficulty of the task, and it’s time to come up with an approach that might surpass the baselines.

Our idea will be to compute cosine similarities using tf-idf representations of n-grams derived from the full names.

Let’s piece this together. This work should be implemented in a scoring method called `tfidf_matcher`, just as you’ve done above.

- To start off, we don’t want to use full names as the tokens. Instead, we’ll be working over character n-grams from the full names. What ideas have we learned that you can use to explain why this is the case?
- After importing a tf-idf vectorizer with `from sklearn.feature_extraction.text import TfidfVectorizer`, you should create a vectorizer object with `vectorizer = TfidfVectorizer(analyzer='char', ngram_range=ngram_range)`. Read the [scikit-learn documentation](#) for the tf-idf analyzer and allow the user to provide input for the `ngram_range` parameter. The default `ngram_range` should support n-grams that vary from 1- to 4-length n-grams. So, here you are creating tf-idf representations based on n-grams for each name.
- Now, the score for this classifier is calculated by taking the cosine similarity between the tf-idf representation of the n-grams of the query with that of each index name. This gives a cosine similarity score of a query name for each index name.
- You should perform some sanity checking in your code. For example, check whether the `ngram_range` argument provided to your `TfidfVectorizer` is within the range described in (b). If it is not, you should raise a `ValueError`.
- Determine an appropriate threshold applying the same work you completed in Part (E). Write that threshold in `discussion.md`.
- Finally, determine the precision, recall, and F-1 at this threshold value.

- (H) Add four `pytest` tests to assess the scoring functions you developed (exact match, Jaccard, Levenshtein, and cosine similarity). For the simple test cases you come up with, store the names in a `test_names.tsv` (or similar) file within the data directory under `tests`. In a larger project, recall, we would add testing code for other parts of the pipeline, too.

- (I) Add a few helpful logging messages. For instance, write logging messages for the beginning and end of the scoring, as well as to report once scoring has completed for every handful of name pairs. You should print enough messages so that your logging is informative to the user, but not so much that it inundates the user with too many logs. Your API (see the example above in Part (F)) does not need to take a logging flag, since we will just treat these messages as the minimally-useful ones for running the application. **Add support to your logging so that you log outputs to a file called `logs.txt` in the root directory of your project.**

- (J) Finally, if you haven’t done so already, make sure you write an informative README demonstrating how to use your code, as well as fill in the remaining files (e.g., `setup.py`, `.gitignore`, etc.) just as we’ve done before.

- (K) Imagine the problem had been a little different. What if, instead, your task was to develop a fuzzy name matching approach that used English and Chinese query names to search over a large index of English and Chinese aliases? The code below pulls English and Chinese politician names, emulating our query list generation.

```
SELECT DISTINCT ?personLabel
WHERE {
  ?person wdt:P106 wd:Q82955;
          rdfs:label ?personLabel.
  FILTER (lang(?personLabel) IN ("en", "zh"))
}
```

LIMIT 250

Take a look at some of the data. You might not be able to glean much, especially if you don’t read Hanzi (Chinese characters). Still, is there anything you do notice about the data? There might not be—and that’s okay. If the dataset had been these English-Russian paired names, write a paragraph in `discussion.md` about what you might do to bridge the language barrier.

- (L) [Bonus] For this bonus, consider the following.
- (a) [2 points] Try to come up with another approach that does even better than the approach in Part (G). It doesn't have to outperform the cosine similarity method, but the method you include should be meaningful nonetheless. Make sure you describe that work in `discussions.md`. I would expect this to involve looking into a technique from the literature and implementing it here. Compare the results with the other methods and place a row in your results table for this approach, as well.
  - (b) [3 points] Use the `multiprocessing` library to parallelize the name pair comparisons. Time your code with and without parallelization and discuss the improvement in `discussion.md`. Make sure you also describe the specs of your machine.

**Grading Rubric.**

Problem	Maximum Score	Your Score	Comments
Problem 1	6		
Bonus	5		
Total	6 (11 with bonus)		

**Grading Guide used by the TA Team.**

- Problem 1: This is a multi-part Python project, so it will likely take some time to grade (4 points).
  - Does the overall project organization look good?
  - Is there an informative README? Do other generic files like the `.gitignore`, `pytest.ini`, and `setup.py` look good?
  - Did they provide good discussions in `discussion.md` for every prompt in the assignment?
  - For the `pytest` tests, do they test for sensible errors?
  - Inside of `name_matcher.py`, do they create a `NameMatchScorer` interface?
  - Does `main.py` use `argparse` and pull together the functionality of the project into one place? For example, do they import their utilities here to process data and then use the main classes to execute the project's main behavior?
  - Do subclasses of the `NameMatchScorer` interface each implement a scoring method?
  - Do students have a function called `evaluate` in `eval.py` that works as described in the assignment?
  - Do students split the `annotated-index.tsv` file into an 80-20 training-test split?
  - Do they have a table listing the precision, recall, and F-1 scores for each approach?
  - Was the Jaccard similarity logic written correctly?
  - Was the Levenshtein distance logic written correctly?
  - Part (G) has students move beyond the baseline approaches. Do they do subparts (a)-(f) thoughtfully and carefully?
  - Do students use logging? Do they run their system and output logs to `logs.txt`?
  - For the extra credit in Part (L), full points are given to each subproblem if students have written working code and provided the requested explanations in their discussion file.