We will begin by discussing the terms of what my project is essentially trying to do, which is a search opinion engine using Amazon reviews. The purpose of this is to match queries with connotations in an accurate setting. We have pre-given data in multiple formats, but I chose to use Python and the pickle file. We are supposed to support three different methods: the baseline method that's based off boolean concepts; method 1, which is my proximity method, in which essentially we choose opinions and aspects that are close in proximity; and the sentiment method, which uses review ratings to help with the connection of the actual sentiment of the review and how it relates to the query. All will be expanded

Beforehand I wanted to mention that in my previous summer software engineering internship, I actually did quite a lot of work with Python and data using a lot of the same libraries and general ideas, so it was actually a really nice project to do, as I got to use a lot of my tools and skills that I developed.

For this we simply need the modules pandas, os, and re. Pandas is the main module here, as it does most of that data manipulation, filtering, etc. re is used for cleaning our text, as it was kind of messy at its original point, and os was used to ultimately print out the outputs of the IDs that passed the criteria of the methods according to their query.

The baseline method works with three tests. The first one, aspect term retrieval, just checks if the review has the aspect word; no opinion term is needed. The second, aspect and opinion match, means the review has to mention both aspect and opinion terms somewhere. The third, aspect or opinion match, is more broad; it accepts reviews

that mention either the aspect or the opinion term. These were given in the instructions, so it was pretty straightforward to get them working, as most of this stuff can be handled with built in pandas functions.

```
    def test(df, queries):

        results = {}

        for key in queries:

            aWord, oWord = queries[key]

            aWordSet = set(aWord)



            test1 = df["tokens_set"].apply(lambda tokens:
len(tokens.intersection(aWordSet)) > 0)

            test2 = df["tokens_set"].apply(lambda tokens:
(len(tokens.intersection(aWordSet)) > 0) and any(word in tokens for word
in oWord))

            test3 = df["tokens_set"].apply(lambda tokens:
(len(tokens.intersection(aWordSet)) > 0) or any(word in tokens for word in
oWord))



            results[key] = {

                1: df.loc[test1, "review_id"].tolist(),

                2: df.loc[test2, "review_id"].tolist(),

                3: df.loc[test3, "review_id"].tolist()

            }
```

```
        return results
```

Method 1, aka the proximity method, built on top of that, with the simple idea of, what if these words were close to each other? Obviously we have to determine how far apart they are and how that will be coded. Obviously you can't predict every scenario, but a maximum window of five words apart felt comfortable to where I could see this could be relevant; for example, "this audio, while talked about, is rather poor." Now time complexity I don't think is super taken into account in grading; however, I did think this was a good opportunity to involve my recently developing LeetCode skills, specifically two pointers, while we could use a more brute force solution to achieve this, I believed it was simple enough to implement. I simply got the positions of each opinion and ascent position (to that specific query and row), and if it was within a 5-word window, they passed the criteria. If not, the l and r pointers iterate according to the situation. This quicker method actually did save time, as when finalizing the results, I did have to run it quite a bit.

```
def proximity(tokens, aWord, oWord, maxi=5):

    aPos = [i for i, token in enumerate(tokens) if token in aWord]

    oPos = [i for i, token in enumerate(tokens) if token in oWord]



    i, j = 0, 0

    while i < len(aPos) and j < len(oPos):

        aCur = aPos[i]
```

```
        oCur = oPos[j]
        if abs(aCur - oCur) <= maxi:
            return True
        if aCur < oCur:
            i += 1
        else:
            j += 1
    return False
```

Last but not least, method 2, which is intended to be the most accurate, builds on top of the past 3. The main addition is utilizing the customer_rating_review variable, which essentially means we match the positive and negative rating to the positive and negative connotation of the opinions. So ultimately this is the final method, which in theory should be the most accurate.

```
def sentiment(tokens,aWord,oWord,rating,maxi=5):
    posOp = {"strong","useful","sharp"}
    negOp = {"poor","problem","click"}

    aPos = [i for i, token in enumerate(tokens) if token in aWord]
    oPos = [i for i, token in enumerate(tokens) if token in oWord]
```

```
rating = int(rating)



i, j = 0, 0

while i < len(aPos) and j < len(oPos):

    aCur = aPos[i]

    oCur = oPos[j]

    dist = abs(aCur - oCur)

    if dist <= maxi:

        oSent = tokens[oCur]

        if oSent in posOp and rating >= 4:

            return True

        if oSent in negOp and rating <= 2:

            return True



    if aCur < oCur:

        i += 1

    else:

        j += 1



return False
```

Now how I gathered the results was I used the table given to collect them; I would check a subset of a certain amount of samples (usually scaling to how many retrieved total) from the retrieved samples that were returned for each method for each query and then tally those that were relevant and then calculate the precision rate. I did have some that were really low so I just checked all of the reviews

Now here is the resulting table

Query	Baseline (Boolean)			Method 1 (M1)			Method 2 (M2)		
	# Ret.	# Rel.	Prec.	# Ret.	# Rel.	Prec.	# Ret.	# Rel.	Prec.
audio quality:poor	1595	259	0.16	898	165	0.18	582	106	0.20
wifi signal:strong	201	70	0.34	138	45	0.33	101	48	0.48
mouse button:click problem	2220	88	0.04	583	52	0.09	147	16	0.11
gps map:useful	224	60	0.27	20	8	0.40	14	7	0.50
image quality:sharp	717	258	0.36	131	57	0.44	111	62	0.56

The results show that method 2 generally performed the best, especially for queries like image quality where the aspect and opinion are usually close and the review rating aligns well with the sentiment. For broader or more varied queries like audio quality and mouse button, the precision was lower across all methods, probably because those terms appear in many broader contexts making it harder to determine relevancy. Interestingly, method 1 didn't improve wifi signal much and even had a dip in precision, which might be because the proximity constraint was too strict for that query where aspect and opinion words weren't always near each other. The gps map query showed a clear boost with method 2, suggesting that combining proximity with sentiment rating helps when opinions are tightly linked to the aspect. Overall, the

baseline returned a lot of results but with lower relevance, method 1 added some focus by considering word closeness, and method 2 gave the best balance by also factoring in review ratings to match sentiment, making it the approach with consistently the highest precision rate.

Overall this project was actually quite rewarding and really allowed me to think in a way the homework hasn't. I think implementing this subtopic in an actual project and using what we have learned taught me a lot about thinking around the limitations of a computer. Looking back, some changes I would’ve made are for the double-word queries; I could’ve made it consider both instead of one or the other, but that’s the neat part of this project. You can easily make improvements incrementally.