HINTS:

FAQ List:

2. Can we use regression?

I strongly urge you to treat it as a regression problem.

4. The official measure we will use in the final evaluation is unbounded NDCG. I will not use a cutoff as there are a different number of documents judged for the queries.

5. I still don't know what the heck is going on. HELP.

Try googling for "the best algorithm for learning to rank". Then google for ALG XXX library. You will find several that will do much of the heavy lifting for you -- I promise. You can start from ground zero and code it all yourself if you want, but I know I wouldn't! You should groupby QueryID using numpy/pandas and drop columns that at not useful when you train, like QueryID and Docid. You need those to generate the final output, but not for modeling. Double hint, if you look around, you will find at least one library that will optimize directly for NDCG! You just need to do a bit of tuning and cross-fold validation to find a good final model to use.

6. Why is this so hard. It isn't fair!

Take a deep breath and reflect on what you want to achieve with a degree in data science -- a job! This is exactly what you will face in the "real world" -- a dataset and a problem. Your job is to find a good solution. You can use any python library you like long as you properly document it and we can reproduce it. Not having the labels for "test" actually isn't a big deal. You *do* have labels for an enormous training set and you can score and tune it easily. The "test" set is simply a holdout set to make sure the model you create actually works for new data. It is trivial to mimic this using what you have available. I know you won't believe me when I say it, but this dataset is *way* cleaner than pretty much anything you'll find in the outside world. The vast majority of data is noisy and has no labels or features defined. You often end up doing that too!

7. Can I submit a jupyter notebook instead of a script?

No. The code does need to move from jupyter to a script and run correctly. I know this is frustrating, but with hundreds of students, it is very difficult to manually run a bunch of juptyter notebooks. It is easy to generate an environment from a requirements.txt and run a script though. The best way to setup your main loop in the python script is to have a flag called "sweep" that if set to true, would run the full sweep. By default, you can leave it set to False where the best parameters get preset and a huge sweep doesn't run. It would then just generate the output file we want (a 3 column tsv file) and

would use the data files named the same (test.tsv, train.tsv) in the same directory. We would save your original run and make sure it was reproducible from the code. If you are unfamiliar with a "main loop", see this code snippet to explain the right way to do it:

```
def parameter_sweep():

    # Something that takes a long time

    return parameters



if __name__ == "__main__":

    sweep = False

    if sweep == True:

        # Set some parameters.

    else:

        parameters = run_paramater_sweep()
```

8. I cannot seem to get GridSearchCV to work with Algorithm X. What do I do?

You have a few choices. The first is to be persistent and get it to work. Finding good parameters is non-trivial with a composite metric like NDCG, and most complex algorithms that have lots of parameters. I have seen good random selection runs where you literally make random choices k times and keep the best one and also ones using something more sophisticated like the hyperopt library, which is a Bayesian metalearning trick that is popular in the ML community now. There are several of these libraries, hyperopt being one of the many, and they tend to be much faster than a brute force sweep of loads of parameters. Brute force is guaranteed to find the best from the set swept, but it is slow. The Bayesian learners tend to find good local minima quickly in a huge space of possibilities. The point is you don't have to use GridSearchCV. You can use anything that is python that you like as long as it is reproducible, ie. a "similar" answer, given that some methods are nondeterministic, and may not find the global optimal solution. If it is clear you know what you did and why you did it and it runs, we'll be happy with it. We just want you to spend you time learning something new that will hopefully be valuable to you in the future!

There are good libraries that are rankers and that's really a groupwise regression, i.e. grouped by query, which is what you'd expect. You generate a top-k list for each query in the end anyway. The key point that is hard to wrap your head around here is that while documents are indeed judged 0,1,2, the goal is to get the 2s to the top, then the 1s, then the 0s. This is best accomplished with floats (regression or scored) as then you are kind of sorting by most likely to be right and not making a fixed decision with a category prediction. It is extremely hard to guess on a document by document basis it's exact relevance label, but generally we can get a pretty decent ordering from most to least likely.

The key point and the hard thing to wrap your head around is that it isn't about the labels in the end -- it is about the best ordering of labels scored as NDCG -- but you can't have an NDGC score for a single document, it is for a single query, regardless of the number of documents labeled in that grouping. It is also extremely hard to optmize directly for NDGG. you can do it "sort of" but the math shows it is not a "differentiable function", so it is not an easy optimization. That said, people *have* figured it out and there are nice clean python APIs you can easily find and use to to do all the heavy lifting for you.

I will be scoring the solutions using exactly that code i.e. the trec_eval implementation of NDCG which may give a different answer than another one. You can dump your validation predictions out from folds and score them using that. Instead of the average, I may give the score for the "best validation fold" which is the one that is usually selected when you run any sort of parameter sweep. Now of course even that is not directly comparable unless you had used exactly the same query folds. If it isn't obvious you *must* create folds as groups of queries. It doesn't matter if it is 5 folds or 3 or 10, it should be by query. Otherwise, you have queries that are split across the train and the validation, and you will get an inflated score on the fold, but when you apply it to the test, it will probably be overtuned or just plain wrong.

The moral of the story is that you get 3 chances for me to score a run on the test set for you. You send it to me and I'll score it (or tell you if it is broken in a way I can't use it). I don't think giving you the ranges with a "corresponding" score for anything like a validation set will be super meaningful as you cannot easily reproduce that. The holdout set is supposed to be different. It is not crazy different but it is definitely not queries your model has seen before -- and that is what it should be -- which is why it is important to do your train/validation splits no matter how you do them by grouping queries and not just randomly splitting by row.

You only need to worry about test.tsv at the end. When you have a model you believe is good, you use it and this file to do predictions, and write it out to a file. That file would contain 3 columns of data in tsv fomat -- queryID,docID,Predcition