Information Visualization II

School of Information, University of Michigan

Week 4:

- Text visualizations

Assignment Overview

The objectives for this week are for you to:

- Understand how to model a corpus using statistical and visual techniques
- Construct an interactive information visualization for search tasks

The total score of this assignment will be

- Problem 1 (20 points)
- Problem 2 (80 points)

Resources:

- We have created two textual datasets for you. One contains the text from Wikipedia pages related to data mining (algorithms, software, techniques, people, etc.). The second is related to *real* mining (equipment, companies, locations, etc.).

Important notes:

1) Grading for this assignment is entirely done by manual inspection. You will have lots of control over the look and feel of problem 2.

2) When turning in your PDF, please use the File -> Print -> Save as PDF option ***from your browser***. Do ***not*** use the File->Download as->PDF option. Complete instructions for this are under Resources in the Coursera page for this class.

If you're having trouble with printing, take a look at this video (https://youtu.be/PiO-K7AoWjk).

In [1]:
```
import pandas as pd
import altair as alt
import json
import ipywidgets as widgets
import spacy
import math
import numpy as np
import scattertext as st
from sklearn import manifold
from sklearn.metrics import euclidean_distances
from datetime import datetime
sp = spacy.load('en_core_web_sm')
```

In [2]:
```
# some utility classes that will help us load the data in
def lemmatize(instring,title="",lemmaCache = {}):
    parsed = None

    if ((title != "") & (title in lemmaCache)):
        parsed = lemmaCache[title]
    else:
        parsed = sp(instring)

    if (lemmaCache != None):
        lemmaCache[title] = parsed
    sent = [x.text if (x.lemma_ == "-PRON-") else x.lemma_ for x in parsed]
    return(sent)

def generateData(filepath,lemmaCache=None):
    articles = []
    with open(filepath) as fp:
        for docid, line in enumerate(fp):
            doc = json.loads(line)
            doclines = doc['text'].split("\n\n")
            ilineid = -1  # we're going to replace the original lineids so there are no gaps
            for lineid,docline in enumerate(doclines):
                obj = {}
                obj['docid'] = docid;
                obj['title'] = doc['title']
                paraterms = lemmatize(docline,doc['title']+str(lineid),lemmaCache)
                obj['text'] = ' ' + ' '.join(paraterms) + ' '
                obj['tokencount'] = len(paraterms)
                if ('category' in doc):
                    obj['category'] = doc['category']
                if (len(paraterms) > 10):
                    ilineid = ilineid + 1
                    obj['lineid'] = ilineid
                    articles.append(obj)
    return pd.DataFrame(articles)

def loadFile(classname,classpath,maxc=200,lemmaCache={}):
    articles = []
    with open(classpath) as fp:
        for docid, line in enumerate(fp):
            doc = json.loads(line)
            doclines = doc['text'].split("\n\n")
```

```
            obj = {}
            obj['docid'] = docid;
            obj['title'] = doc['title']
            paraterms = lemmatize(doc['text'],doc['title'],lemmaCache)
            obj['text'] = ' ' + ' '.join(paraterms) + ' '
            obj['label'] = classname
            if ('category' in doc):
                obj['category'] = doc['category']
            if (len(paraterms) > 10):
                articles.append(obj)
            if (docid > maxc):
                break
    return(articles)

def loadClasses(class1name,class1path,class2name,class2path,maxc=300):
    articles = loadFile(class1name,class1path) + loadFile(class2name,class2path)
    return pd.DataFrame(articles)
```

In [3]:
```
# enable correct rendering
alt.renderers.enable('default')

# uses intermediate json files to speed things up
alt.data_transformers.enable('json')
```

Out[3]:
```
DataTransformerRegistry.enable('json')
```

Before we start...

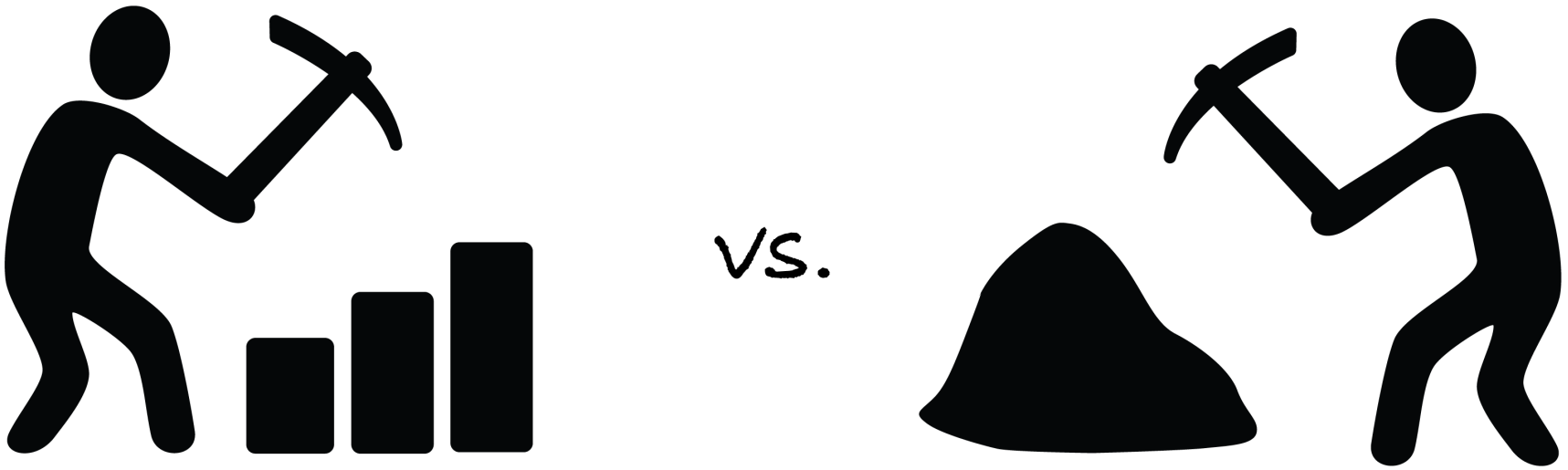
We have created a function for you called `lemmatize(...)` . It takes as input a string and assumes that spaces are token delimiters. For each token/word, the system will lowercase it, stem it (getting the root), and generally clean it up. The data we load from our files undergoes the same transformation. So it's important to lemmatize your terms if you are looking them up. For example, you won't find the word "data" in the DataFrame. All instances get transformed to "datum." Thus, it's important to remember to do this transformation. Note, however, that if you query for "Data Mining" (in capitals), the system assumes that this is a name, and will not change it.

In [4]:
```
# here's a few examples

query1 = "data mining"
print("The lemmatized version of '"+query1+"' is:", lemmatize(query1))

query2 = "Data Mining"  # proper name (like a business)
print("The lemmatized version of '"+query2+"' is:", lemmatize(query2))

query3 = "executing awesome algorithms"
print("The lemmatized version of '"+query3+"' is:", lemmatize(query3))

query4 = "Data mining algorithms are super exciting."
print("The lemmatized version of '"+query4+"' is:", lemmatize(query4))
```

```
The lemmatized version of 'data mining' is: ['data', 'mining']
The lemmatized version of 'Data Mining' is: ['Data', 'Mining']
The lemmatized version of 'executing awesome algorithms' is: ['execute', 'awesome', 'algorithm']
The lemmatized version of 'Data mining algorithms are super exciting.' is: ['data', 'mining', 'algorithm', 'b
e', 'super', 'exciting', '.']
```

Problem 1 (20 points)

For this first problem, we will be comparing which terms most often appear in which of our two corpora: 'data' mining and 'real' mining.



Let's load the data in:

In [5]:
```
# this will load the two files and label them with one of the two class labels
# lemmatizing these files takes some time on Coursera so we've pre-calculated it for you.
# If you want to run this process, uncomment the next two lines of code
# miningdf = loadClasses('data mining','assets/mlarticles.jsonl','real mining','assets/miningarticles.jsonl')
# miningdf.to_csv("assets/miningvmining.csv",index=False)

# load from cached file
miningdf = pd.read_csv("assets/miningvmining.csv")
```

In [6]:
```
# let's look at what's inside.  We have a document id (docid), title (from Wikipedia)
# the text, the label (one of: 'real mining' or 'data mining'), and a category column
# which you can ignore for now (it has the Wikipedia category for just the data mining articles)

miningdf.sample(5)
```

Out[6]:

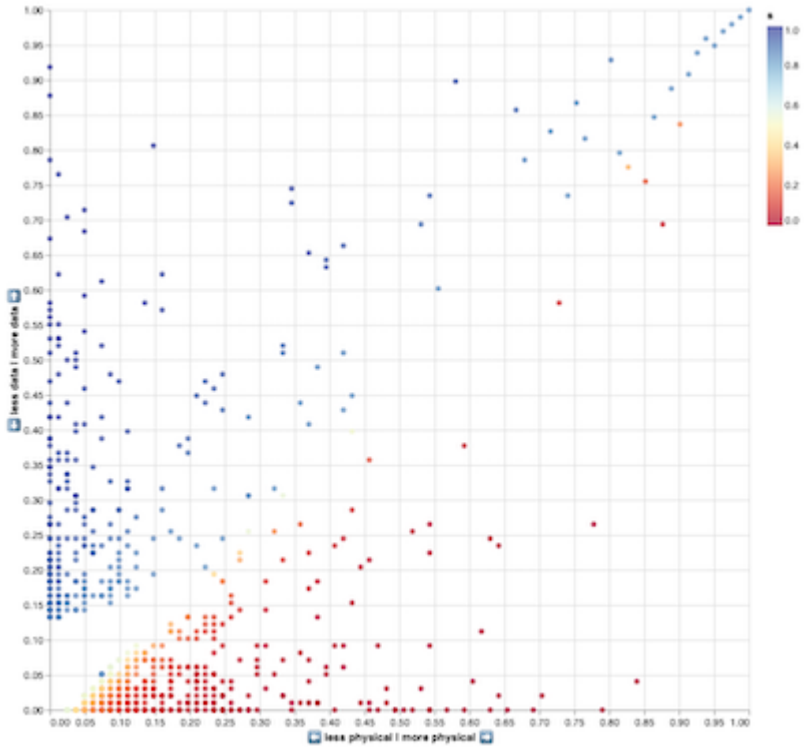
	docid	title	text	label	category
183	183	Learning with errors	in machine learning , a subfield of computer ...	data mining	Machine learning
267	65	Lynn Lake	lynn lake be a town in the northwest region o...	real mining	NaN
341	139	Motru	motru ( ) be a municipality in romania , gorj...	real mining	NaN
396	194	Kinross, Mpumalanga	kinross be a small gold mining town in mpumal...	real mining	NaN
180	180	Yann LeCun	yann lecun (; originally spell le cun ; be...	data mining	Machine learning researchers

In [7]:
```
# you'll notice that the text is lemmatized. Here's the text for the first entry:
miningdf.head(1).text.values[0]
```

Out[7]:
```
' matlab ( matrix laboratory ) be a multi - paradigm numerical computing environment and proprietary programmi
ng language develop by mathworks . matlab allow matrix manipulation , plot of function and datum , implementat
ion of algorithm , creation of user interface , and interfac with program write in other language , include c
, c++ , c # , java , fortran and python . \n\n although matlab be intend primarily for numerical computing , a
n optional toolbox use the mupad symbolic engine , allow access to symbolic computing ability . an additional
package , simulink , add graphical multi - domain simulation and model - base design for dynamic and embed sys
tem . \n\n as of 2018 , matlab have more than 3 million user worldwide . matlab user come from various backgro
und of engineering , science , and economic . '
```

We will be using the scattertext (https://github.com/JasonKessler/scattertext) library to create, analyze and position the terms. We encourage you to take a look at all the features of scattertext. It has a lot of "knobs" to control the analysis. It will also create a very fancy Web-based, interactive visualization for you if you want. For our initial experiment, we're going to start simple. We simply want to plots

terms based on how common they are in 'data mining' and in 'real mining.' The lower-left corner will hold uncommon terms for both. The upper right will be terms that often appear in both domains. A way to think of this is that terms on the diagonal (slope 1) appear equally in both domains. The other two corners are the outliers--these are terms that are either more common for data or real mining. Here's a screenshot of what we'll get:



Click here (assets/scattertext.png) for a larger image.

We're going to run the analysis for you and have you generate the visualization. Once you get the first version working, you can play with the options to see how they impact the analysis/visualization. In particular, you might want to change the term frequency and PMI (pointwise mutual information) thresholds to see what they do.

In [8]:
```
# apply the scattertext analysis pipeline to the text, this will create a new column called parse
miningdf = miningdf.assign(
    parse=lambda df: df.text.apply(st.whitespace_nlp_with_sentences)
)

# create a "corpus" object
corpus = st.CorpusFromParsedDocuments(
    # use the miningdf as input. The category col is "label" and the parsed data is in "parse"
    miningdf, category_col='label', parsed_col='parse'
    # the unigram corpus means we want single words (there's another version that throws out stopwords)
    # the association compactor says we want the 2000 most label-associated terms
).build().get_unigram_corpus().compact(st.AssociationCompactor(2000))

# next, we build the actual visualization
scatterdata = st.produce_scattertext_explorer(
    corpus,                              # the corpus
    category='data mining',              # the "base" category
    category_name='data mining',         # the label for the category (same in this case)
    not_category_name='real mining',     # the label of the other category
    minimum_term_frequency=0,            # threshold frequency
    pmi_threshold_coefficient=0,         # the PMI threshold
    return_data=True,                    # this tells scattertext to return the data rather than saving an HTML pa
    transform=st.Scalers.dense_rank      # where to place identically ranked terms (on top of each other here)
)
```

At this point, `scatterdata` will contain all kinds of information. For example, `scatterdata['info']['category_terms']` will give you the terms most related to the "category" (remember, this is data mining). In contrast, you can get the "real mining" terms using not_category_terms.

In [9]:
```
print("terms most associated with data mining ",scatterdata['info']['category_terms'],"\n")
print("terms most associated with real mining ",scatterdata['info']['not_category_terms'])
```

```
terms most associated with data mining  ['datum', 'learning', 'algorithm', 'analysis', 'computer', 'data', 'mo
del', 'machine', 'research', 'pattern']

terms most associated with real mining  ['mine', 'town', 'gold', 'south', 'coal', 'locate', 'miner', 'diamon
d', 'river', 'north']
```

The more important piece for our visualization purposes is the "data" part of scatterdata. This is a list of "objects," one for each term. For example:

```
scatterdata['data'][0]
```

```
{'x': 0.0,
 'y': 0.13265306122448978,
 'ox': 0.0,
 'oy': 0.13265306122448978,
 'term': 'matlab',
 'cat25k': 15,
 'ncat25k': 0,
 'neut25k': 0,
 'neut': 0,
 'extra25k': 0,
 'extra': 0,
 'cat': 13,
 'ncat': 0,
 's': 0.8930722891566265,
 'os': 0.12921901946292189,
 'bg': 1.1352105640948616e-05}
```

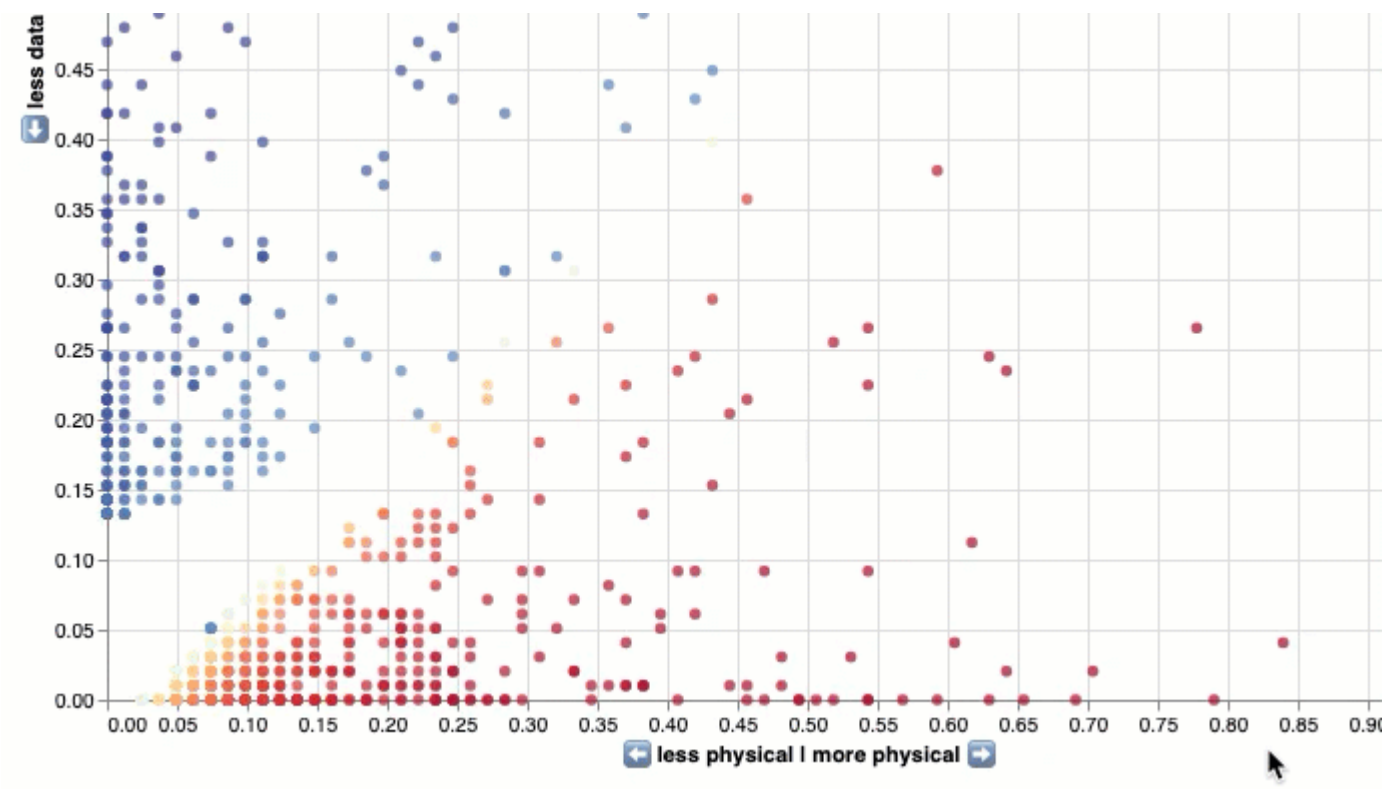
This is the first item in the data list. There are a number of fields here. You can look at the documentation for scattertext for the details. The only items we care about right now will be `x`, `y`, `term`, and `s`. These respectively tell us the x/y coordinate for the term, the term itself, and the "distance" of the term from the central line (slope 1).

In [10]:
```
scatterdata['data'][0]
```

Out[10]:
```
{'x': 0.0,
 'y': 0.13265306122448978,
 'ox': 0.0,
 'oy': 0.13265306122448978,
 'term': 'matlab',
 'cat25k': 15,
 'ncat25k': 0,
 'neut25k': 0,
 'neut': 0,
 'extra25k': 0,
 'extra': 0,
 'cat': 13,
 'ncat': 0,
 's': 0.8930722891566265,
 'os': 0.12921901946292189,
 'bg': 1.1352105640948616e-05}
```

We would like for you to use this data to generate a visualization as in the example above. You're welcome to try to make it fancier, but consider this the minimum solution (notice the tooltips and colors).

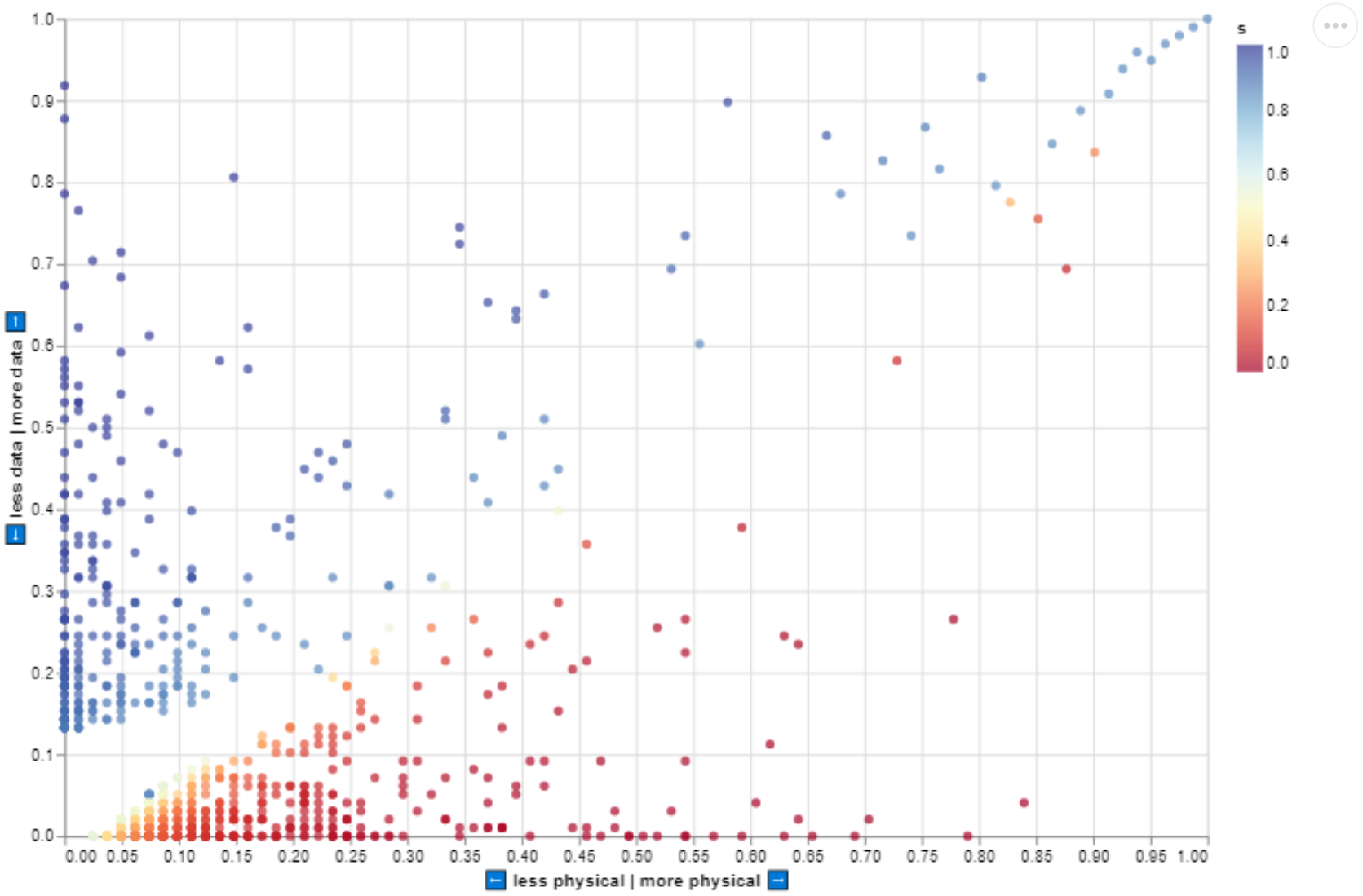
Here's a *zoomed in* version just to show off the interactions:



In [11]:
```
# add your solution here

def genScattertext():
    # this function should return an Altair chart as specified above
    df = pd.DataFrame(scatterdata['data'])

    toret = alt.Chart(df) \
                .mark_circle() \
                .encode(x = alt.X('x', title = '⬅ less physical | more physical ➡'),
                        y = alt.Y('y', title = '⬅ less data | more data ➡'),
                        tooltip = 'term',
                        color = alt.Color('s', scale = alt.Scale(scheme = 'redyellowblue'))
                       ) \
                .properties(width = 700, height = 500)

    return(toret)
```

In [12]:
```
# if your code above works correctly, this should generate the plot
genScattertext()
```

Out[12]:



Problem 2

Problem 2.1 -- Implementation (70 points)

For this problem, we would like for you to build a visual query system using tilebars! You will need to build a function that returns an Altair visualization. It will take as input a query (text string), an option to normalize the data or not (A boolean True or False), and a string indicating the sort order--"title" (or "name") and "score". Here's a pretty bad implementation:

```
searchbutton = widgets.Button(description="Search")
normalizedradio = widgets.RadioButtons(description="Normalized?",options=['true', 'false'])
sortradio = widgets.RadioButtons(description="Sort by",options=['name', 'score'])

searchbutton.on_click(clicked)
normalizedradio.observe(clicked)
sortradio.observe(clicked)

list_widgets = [widgets.VBox([widgets.HBox([querybox,searchbutton]),
                              widgets.HBox([normalizedradio,sortradio])])]
accordion = widgets.Accordion(children=list_widgets)
accordion.set_title(0,"Search Controls")
display(accordion,output)
```

]:

Important: This example implementation isn't a great one. We expect you to use your skills to build something better. Simply replicating this example will not get you full credit.

You are welcome to come up with your own style, add interactivity, decide how to normalize and score, the data, etc. This problem is very open ended. If you don't remember how a tilebar is created, now is a good time to go back to the lecture and watch the video (or go to the source (https://people.ischool.berkeley.edu/~hearst/research/tilebars.html)).

Before we get started, let's load the corpus:

In [13]:
```
# we're only working with data mining here


# lemmatizing these files takes some time on Coursera so we've pre-calculated it for you.
# If you want to run this process, uncomment the next three lines of code
# We're going to "cache" lemmas to speed up some operations
# lemmaCache = {}
# dataminingdf = generateData('assets/mlarticles.jsonl',lemmaCache)
# dataminingdf.to_csv('assets/mlarticles.csv',index=False)

dataminingdf = pd.read_csv('assets/mlarticles.csv')
```

In [14]:
```
# let's look at the first few lines
dataminingdf.head(5)
```

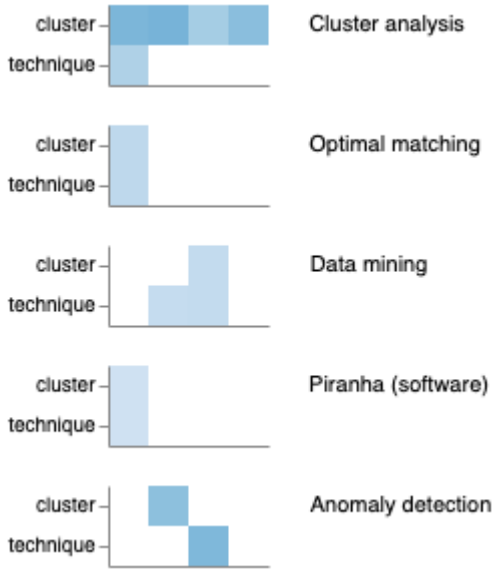
Out[14]:

	docid	title	text	tokencount	category	lineid
0	0	MATLAB	matlab (matrix laboratory) be a multi - par...	64	Data mining and machine learning software	0
1	0	MATLAB	although matlab be intend primarily for numer...	48	Data mining and machine learning software	1
2	0	MATLAB	as of 2018 , matlab have more than 3 million ...	27	Data mining and machine learning software	2
3	1	Ray Kurzweil	raymond kurzweil (; bear february 12 , 1948 ...	105	Machine learning researchers	0
4	1	Ray Kurzweil	kurzweil receive the 1999 national medal of t...	141	Machine learning researchers	1

What you see above is a row for every document and every "line." In pre-processing the top section of each Wikipedia article for you. We have taken each paragraph and made it into a new line. For example, the MATLAB (https://en.wikipedia.org/wiki/MATLAB) article has 3 lines. The frame has a document id (docid), title, line id (lineid -- based on the order the line appears), text (the text of the line), tokencount (the number of words in that line), and the category of the article (which you can use if you want).

At this point we're going to start implementing the `drawTilebars` function.

As we mentioned above, everything outside of the basic functions and tilebar encoding is fair game. You can decide how to rank documents (do you want to do it based on whether the matches are in the same line? whether there are many matches throughout the article?). You can also decide how you want to implement normalization on the tilebars themselves (by tokens in the line? by the maximum times the term appears in the document? the maximum time it appears in all documents?). Here's a static screenshot of our tilebars for "clustering techniques" normalized with score based ordering:



Some hints/ground rules (***read before you start***):

- ***Again, this particular version is NOT a good version, please don't simply replicate it.***
- We would like results to be *conjunctions* (i.e., we only want results to show those documents that match *all* search results: basically "ands). You can *optionally* implement disjunctions (i.e., "ors") and modify the interface to support that. You can extend the widget/function to support this if you like.
- Decide what comparisons need to be enabled by tilebars for them to work. Figure out how those can be *expressed* in the visualization and then made *effective*. There are some very simple things that will take our bad example and make it better. You should go back to the video/other materials. We will expect that *at least* satisfy the key tasks of the original tilebar implementation but you can add your own.
- We will consider your design choices in your grade. How it performs will matter, but so will how it looks.
- Make sure that sort by score does something reasonable. Think about what you would expect to be on top when you search for 'cluster analysis' (for example)
- Test, test, test - small queries, big queries, queries with no matches, etc
- Your solution should be performant. We should not be waiting more than a few seconds for the vis to show up.
- Take a look at some of the pandas features for text analysis. For example, for a row in our dataframe, you can get the count of the number of times a specific token appears by doing `row['text'].count(' ' + term + ' ')`
- You likely want to calculate two things--one is a dataframe describing your tilebar information, the other is some kind of document order. If you're clever, you can do it all at once. A less efficient solution might require two passes.
- Think about what you need to know in order to encode the "cell" of the tilebar. Your dataframe should contain that data.
- Consider the look and feel of your solution. We will be considering the aesthetic choices you are making.

- Think through how to build the "small multiples" here. You can use combinations of concatenation, faceting, and repeated charts. You'll likely need to play with a few solutions to get the look you're happy with.

In [15]:

```
#Build DataFrame used in function first

#get total paragraphs
dataminingdf['totaldocs'] = len(dataminingdf)

#put words and punctuation from the paragraph into a list in the DataFrame column
dataminingdf['list'] = dataminingdf['text'].apply(lambda x: x.split())

#explode the list into separate rows
explode = dataminingdf[['totaldocs', 'docid', 'title', 'lineid', 'list']].explode('list')

#use to find frequency in the group by
explode['freq'] = 1

#find frequency of a word or punctuation in the paragraph
grouped = explode.groupby(['totaldocs','docid', 'title', 'lineid', 'list'])['freq'].sum().reset_index()

#use the dummy value in the group by as part of the normalization process
grouped['temp'] = 1

#find frequency of a word appearing in ALL paragraphs
total_appearances = grouped.groupby(['list'])['temp'].sum().reset_index()

#merge in the total appearance data to the base DataFrame
merged = grouped.merge(total_appearances, how = 'left', on = 'list')

#calculate a normalized frequency based on TFIDF
merged['freq_normed'] = np.log10(1+ merged['freq']) * np.log10(merged['totaldocs']/merged['temp_y'])

#pull through columns of interest
freq = merged[['docid', 'title', 'lineid', 'list', 'freq', 'freq_normed']]

#find the total number of paragraphs in document to adjust scales later
temp_merge = freq.groupby(['docid'])['lineid'].nunique().reset_index()

#merge in total paragraphs and create the final DataFrame
final = freq.merge(temp_merge, how = 'left', on = 'docid')
final.rename(columns={'lineid_x': 'lineid', 'lineid_y': 'paragraphs', 'list': 'word',
                      'freq': 'paragraph_frequency', 'freq_normed': 'normalized_paragraph_frequency'}, inplace =

def drawTilebars(query,normalized=False,sortby='title'):
    # this function takes
```

```
# query: a string query
# normalized: an argument about whether to normalize the tilebar (True or False)
#   if false, the the color of the tile should map to the count
#   if true, you should decide how you want to normalize (by the max count overall? max count in article?)
# sortby: a string of either "title" or "score"
#   if title, the tilebars should be returned based on alphabetical order of the articles
#   if score, you can decide how you want to rank the articles
# the function returns: an altair chart
print("the lemmatized query terms are: ",lemmatize(query))
print("normalized is ",normalized)
print("I will sort by", sortby)

#split query based on space delimiter
split_query = query.split()
len(split_query)

#search the DataFrame created above for the words found in the query
#all words in the query must be found within a doc for the doc to qualify
restricted = final[final['word'].isin(split_query)]
in_doc = restricted.groupby(['docid'])['word'].nunique().reset_index()
in_doc = in_doc[in_doc['word'] == len(split_query)]

#limit the DataFrame to the qualifying docs that have all words contained in the query
valid_docs = list(in_doc['docid'])
restricted = restricted[restricted['docid'].isin(valid_docs)]

#create a separate DataFrame that has the score values for each doc based on query term frequency (used for
scores = restricted.groupby(['docid', 'title', 'paragraphs'])['paragraph_frequency', 'normalized_paragraph_f
               .sum().reset_index()

#raw and normalized scores are further weighted by the number of paragraphs in the document
scores['freq'] = scores['paragraph_frequency'] / scores['paragraphs']
scores['normed_freq'] = scores['normalized_paragraph_frequency'] / scores['paragraphs']

#if the tilebar is Normalized = True, use the normalized paragraph frequency, set the color scheme to reds a
#by the normalized frequency score generated above. If False, use the paragraph frequency, set the color sch
#and sort by the raw frequency score generated above
if normalized:
    value = 'normalized_paragraph_frequency:Q'
    scheme = 'reds'
    score_sort = list(scores.sort_values(by = ['normed_freq'], ascending = False)['title'])
else:
    value = 'paragraph_frequency:Q'
```

```
        scheme = 'blues'
        score_sort = list(scores.sort_values(by = ['normed_freq'], ascending = False)['title'])

    #If-else statement that will allow for sorting within the tilebar either by title or the scores generated ab
    if sortby == 'title':
        docs = list(restricted['title'].unique())
        docs = sorted(docs)
    else:
        docs = score_sort

    #for loop that generates each row of the small multiples plot that is eventually generated through a vconcat
    #iterates through each of the qualifying docs and uses the number of paragraphs in the doc to adjust the wia
    #The plot itself uses a rectangle mark and encodes the paragraph (lineid) on the x axis, with the axis value
    #ranging from 0 to the number of paragraphs in the document. The query words are encoded on the y-axis. Ther
    #that displays the document title, the paragraph number, the search word, and either the paragraph frequency
    #paragraph frequency depending on whether normalized is true or false. The color of the plot is also encodea
    #whether normalized is true or false and the color scheme additionally corresponds to this binary value. Las
    #is encoded by the title of the document.
    charts = []
    for doc in docs:
        iter_df = restricted[restricted['title'] == doc]
        paragraphs = list(iter_df['paragraphs'].unique())[0]

        charts.append(alt.Chart(iter_df).mark_rect() \
                        .encode(
                                x=alt.X('lineid:N', title = '',
                                        axis=alt.Axis(values=[i for i in range(paragraphs+1)], labelAngle = 0),
                                        scale=alt.Scale(domain=[i for i in range(paragraphs+1)])),

                                y=alt.Y('word:N', title = ''),
                                tooltip=['title:N', 'lineid:N', 'word:N', value],
                                color= alt.Color(value, legend = None, scale = alt.Scale(scheme = scheme)),
                                row = alt.Row('title', header=alt.Header(labelAngle=0, labelAlign = 'left'), tit
                                ) \
                        .properties(width = 100*paragraphs, height = 100))

    return alt.vconcat(*charts)
```

If you built your solution correctly, you should be able to simply run the code below. Note that we don't use Altair interactivity because we don't know how you chose to implement your solution. The visualization will likely flicker as you recalculate it.

In [16]:

```
output = widgets.Output()
from IPython.display import display

lastquery = ""
lastsort = ""
lastnorm = ""


def clicked(b):
    output.clear_output()
    with output:
        _norm = True
        _sortby = 'title'
        _query = querybox.value

        if (normalizedradio.value == "false"):
            _norm = False

        if (sortradio.value == 'score'):
            _sortby = 'score'

        if (_query == ""):
            print("please enter a query")
        else:
            drawTilebars(_query,normalized=_norm,sortby=_sortby).display()


querybox = widgets.Text(description='Query:')
searchbutton = widgets.Button(description="Search")
normalizedradio = widgets.RadioButtons(description="Normalized?",options=['true', 'false'])
sortradio = widgets.RadioButtons(description="Sort by",options=['title', 'score'])

searchbutton.on_click(clicked)
normalizedradio.observe(clicked, names=['value'])
sortradio.observe(clicked, names=['value'])

list_widgets = [widgets.VBox([widgets.HBox([querybox,searchbutton]),
                              widgets.HBox([normalizedradio,sortradio])])]
accordion = widgets.Accordion(children=list_widgets)
accordion.set_title(0,"Search Controls")
display(accordion,output)
```

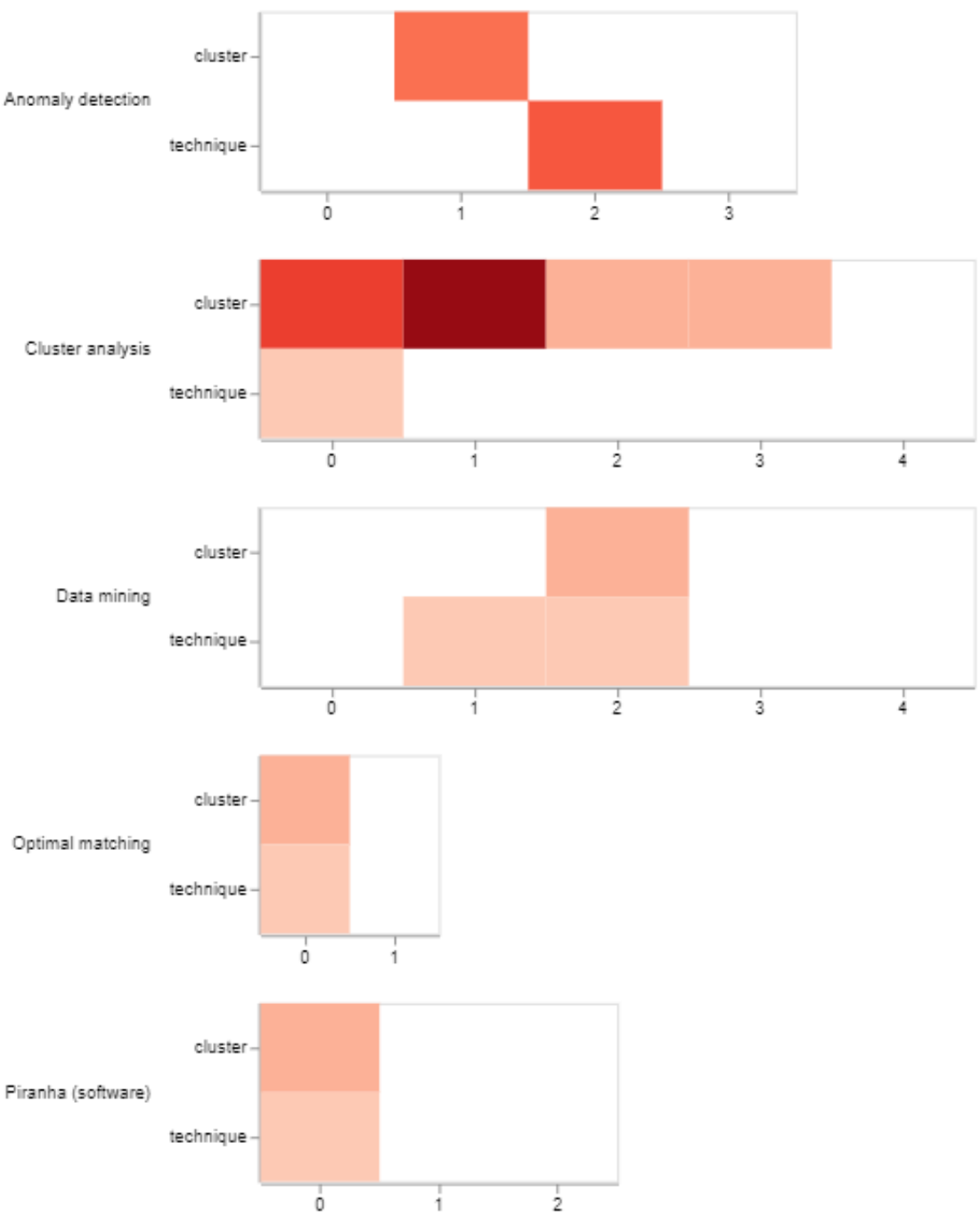
Search Controls

Query: cluster technique Search

Normalized? true false

Sort by title score

```
the lemmatized query terms are:  ['cluster', 'technique']
normalized is  True
I will sort by title
```


Anomaly detection
cluster
technique
0
1
2
3
Cluster analysis
cluster
technique
0
1
2
3
4
Data mining
cluster
technique
0
1
2
3
4
Optimal matching
cluster
technique
0
1
Piranha (software)
cluster
technique
0
1
2

Problem 2.2 What did you do? (10 points)

Please detail why you made your design decisions. Again, we want you to improve on our less-than-optimal implementation of tilebars. You should reflect on how well you are meeting each of the objectives of tilebars.

The process for generating the tilebars and how the normalization was performed, how the sort functionality was implemented, how the various documents were filtered, etc. is all detailed within the annotated code above. In terms of a comparison between the created visualization above and the less-than-optimal implementation, the above visualization is more expressive and effective and meets all three tilebar objectives noted in this week's lecture.

The first objective of a tilebar is to indicate relative document length. The less-than-optimal implementation did not have axis labels and had the x-axis length for all of the qualifying documents as the same size. For the above visualization, x-axis labels are present and denote the number of paragraphs in each of the qualifying documents. Further, the x-axis length is adjusted based on the number of paragraphs in the document. Documents with more paragraphs will have longer tilebars than those with only a couple of paragraphs. The above visualization is therefore more effective than the example implementation as the document length can easily be determined based on the x-axis labels and the relative document length can easily be determined based on x-axis width.

The second objective of a tilebar is to determine the frequency of term sets in a document. The above visualization clearly displays each term in the search query on the y-axis and its prevalence in each of the paragraphs for documents that contain ALL of the search terms. This visualization encodes the word frequency, whether or not it's raw or normalized, based on the color, but the tooltip functionality that displays the document name, paragraph number, the word, and its raw or normalized frequency is also a more effective way of displaying this frequency relative to the less-than-optimal implementation. The less than optimal implementation just relies on a gradient color scale to denote relative frequency. The above visualization also encodes this gradient color scale but further provides detail around the specific scores, leading to both a more effective and expressive visualization.

The third and final objective of a tilebar is to determine the distribution of term sets with respect to the document and each other. Similarly to the second objective, this is well handled by the above visualization. Whether the score is raw or normalized, a color gradient encodes its frequency across the paragraphs in a document and across documents, and a tooltip allows for a more effective visualization than the sample as the exact frequency can more easily be determined. Overall, the three objectives of a tilebar are successfully implemented by the above visualization which improves significantly on the example implementation.

In [ ]: