Project 6: Deep dive

This is finding little problems. It is the second half of the 80/20 rule. The marginal gains are small compared to the timed being invested.

Problem: The results of project 6 were not quite ready for prime time.

This project is to prepare the code for use in other contexts, api calls and life on a server.

Downloading from Wikipedia:

**Problem 1. API calls are limited to 500 reponses. **

Solution. Making the function for reading the pages in a category recurrsive when there is a kwarg 'continue_string'. The recursive call is obsfucated to prevent it being called accidentally. Status: resolved. See notebook 'Problem1.ipynb'

Problem 2. Service architecture.

Solution: Storing the data requires a database service. Local Storage through was done through postgres. The class project used a hosted service. To persist this project, it needs a dedicated service. Also, tranportability/host independence is desirable for minimizing costs.

Separating the data store from the database host is also desirable. This will make migrations and backups easier. Configuring the service will be harder.

I've chosen Docker as the first iteration of containers to use for abstracting from the hosting architecture. Alternatives exist but do not appear to be as widely adopted.

Docker is a living a growing system. Changes to the functionality should be expected and the internet is a fabulous source of outdated information. Previous versions of docker utilized data only containers. Much of the documentation in the wild refer to these structures. Current best practice is using a named container linked to a physical persistasnt data store. This named container shares its data volumes to other containers at run time.

Status: incomplete. but serviceable.

Problem 3. Postgres in Docker.

My experince configuring postgres with a persistant data store was very challenging. There are 2 critical areas to store: the data and the configuration for the database. The base image for postgres was problematic for me. It either coulldn't reloaded the data from a persistant store, or it could overwrote the configuration of the server with the default configuration.

Using Ubuntu images on AWS, this code allowed me to persist the data beyond the the instance on EC2.

```
sudo docker create -v /mnt/postgres/postgres_files/ --name postgres_aws library/postg
res /bin/true
sudo docker run -p 5432:5432 -e POSTGRES_PASSWORD=aPassword -d --volumes-from postgre
s_aws postgres
```

Source

The element that was most clearly missing for me was using `--volumes-from`. The mechanism where the volume is mounting and how it works is opaque to me.

To Do: roll my own image with external volume and utilize docer compose.

Problem 4. Acquiring data

For the most part, this is has been resolved. I have a nifty command line tool that will read a yaml file or arguments and try to search for those as categories. Also, the comand line tool can write to different database locations, like the local computer or AWS.

The main problem with the data acquisition is that is it is slow. The command tool doesn'tl report it's progress or estimated time to completion. I've added a progress bar. It called tqdm. I highly recommend it. It decorates any iterator and creates a progress bar in a notebook, terminal or a gui. It also estimates the time to completion.

Problem 5. Network bottleneck.

Downloading data is boring and slow. Each api call has a tremendous overhead/latency. The downloading process is really a `for` loop which contains a function call and a prameter from a list. There is very boring process could be run in parallel, rather than serial.
To convert to a parallel process, another level of abstraction is required. This calls of `redisqueue` `Redisqueue` is a job queue. It takes a function and a set of parameters, then uses a worker to execute the job. Many workers can call from a queue. Now the boring downloading process is parallelized.

Status: to implement.

Problem 6. Modeling Cosine Similarity works pretty well. So does everything else: Categories for

Train/Test/Validation:

category_name	# of pages	Validation Category
sports cars	593	muscle cars
Arcade games	1644	Cancelled arcade games
desserts	161	cookies
chemistry	120	Industrial gases
physics	45	physicists
psychology	288	Popular psychology
cat breeds	94	Natural cat breeds
Earth sciences	105	Geochemistry
Sandwiches	122	American sandwiches
Breads	127	Bread dishes
Automotive technologies	163	Vehicle dynamics
belief	72	Ignorance
hygiene	95	Ritual purification
sports terminology	178	martial arts terminology
shoes	92	Sneaker culture
influenza	40	Influenza researchers
Children	6	Sons of Odin
Physical_quantities	232	Density
Submarine sandwich restaurants	42	
Association football trophies and awards	58	
Physical exercise	190	hydrotherapy
health care	85	medical robotics
machine learning	189	Classification algorithms

The models were tested with some classifiers. Each model used the same train/test split. Where it is feasible, a cross validated grid search was done to select model parameters.

Category Prediction Table: without lemmatization, ~4000 documents

model	train	test	out of sample	params	Time to build model	Time to predict one page
Cosine Similarity	94.5%	91.7%	72.3%		18.0 sec	4.1 ms
K Nearest Neighbors	90.2%	89.3%	64.0%		79 ms	1.45 ms
Multi Layer Perceptron	97.7%	94.8%	70.8%	activation= logistic, solver = 'adm', hidden_layer_size = (100,)	26.3 sec	3.7 ms
Random Forest Classifier	99.7%	91.5%	67.3%	estimators = 30	1.6 sec	120 µs
xgBoost	99.7%	92.7%	66.8%	n_estimators = 1000, max_depth = 5	39 min	6.6 ms

to do: re-evaluate with more categories and more pages.

Problem 7. tf-idf features tf-idf results in lots of features. Before cleaning the text, tf-idf gave me 130,000 unique features. Numbers, puncutation and other mark up features contributed many non-word items being included as features. For the moment, I'll deny these non-word things have meaning, and I'll eliminate them as they are found.

First pass for cleaning:

```
temp = page.replace('\n',' ') #remove new lines
temp = temp.lower()
temp = re.sub(r'[^\w\s]',' ',temp) #remove all non-word, non-whitespace characters.
temp = re.sub(r'\[[0-9]{1,10}\]',' ', temp) #Remove strings of nunmbers
```

Stemming and Lemmetization are additional methods that could reduce the features. Both are computational expensive, and did not eliminate a large number of features. Maybe this is worth revisiting later.

Problem 8. Alternate to truncatedSVD We've discussed the use of Neural Nets for use with image processing. There is another problem that has a very large number of features that could be lead to a classification.
Just as a first pass, using a Multi Layer Perceptron Classifier on a tfidf **without** further transformation yielded results that are comparable other classifiers (train: 99.7%, test: 95.5%, Validation: 72.0%).

To Do: test keras with tfidf.

Also, Multinomial bayes failed spectacularly.

Problem 9. Data beyond Wikipedia I've maintained the data structure based upon the original specification of retrieveing data from wikipedia. I've used wikipedia's ids as the primary keys for the category and page tables. To include data from any other source, the data structure will need to expand to include a key field that is based unique to the datastore. And meta data fields will need to be added to include the source of the category and pages.

Problem 10. The footprints Of course, someone has been here and done this before me. Radim Řehůřek has done an unsupervised version, to the point is an example included in gensim. He even has a distributed version

An all of the data acquisition can be bypassed by taking the Data dump straight from Wikipedia.

Problem 11. Multiple catgories Many documents belong to mulitple categories, and categories have parent categories. the categorization of a document should reflect the inclusion of multiple categorizations. Rather than fighting the complexity, this classifer should embrace the chaos and predict multiple possible categories.

to do: add the categories of the pages, not just the category that lead to discovering the page.