Building a sustainable recipe recommender

Created: Aug 2020
Last modified: Oct 2020

Steffen Buergers
@author: sbuergers at gmail dot com

The idea for a sustainable recipe recommender emerged when I was brainstorming possible viable capstone projects for the data incubator, a data science bootcamp for academics who want to become data scientists in the industry. I knew that I wanted to do a project that I personally cared about, showcased my data science skills to employers, and, in a perfect world, would also be useful to others once completed. I considered topics such as climate change, Covid19, policy decisions, energy consumption and health care - all of which had a large amount of data that could be publicly accessible. However, every individual data source, being publicly accesible, was already analyzed at length, and I was struggling to find a viable link between different data sources such that I could combine them in new and interesting ways.

Luckily, I came across an article on ourworldindata.org, by Hannah Ritchie and Max Roser that summarizes and discusses the implications of a 2018 paper published in Science that assesses the life-cycle environmental impact of approximately 90% of the world's food production (Poore & Nemecek, 2018). And both the paper and data are freely available online. After this discovery it was not much a stretch to derive a plan for my capstone project: Build a sustainable recipe recommender based on two datasets: 1.) a recipe database, which can be obtained from any recipe website using web-scraping, and 2.) the tabular greenhouse gas emission data provided by Poore and Nemecek (2018).

In particular, for a minimum viable product, **my goal was to build a web-app where people can look up the associated greenhouse gas emissions of a recipe of their choice, and receive recommendations for similar recipes that are more sustainable.**

Legend

Getting the data

Aug 2020

The dataset about greenhouse gas emissions was ready to be fed into pandas, no preprocessing needed.

In [1]:
```
# import libraries and load dataset

import pandas as pd
import matplotlib as mpl
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import scipy as sci

# download the dataset and substitute the path and filename as needed
df_GHG = pd.read_csv(r'D:\data science\nutrition\GHG-emissions-by-life-cycle-stage_orig.csv')
df_GHG.head()
```

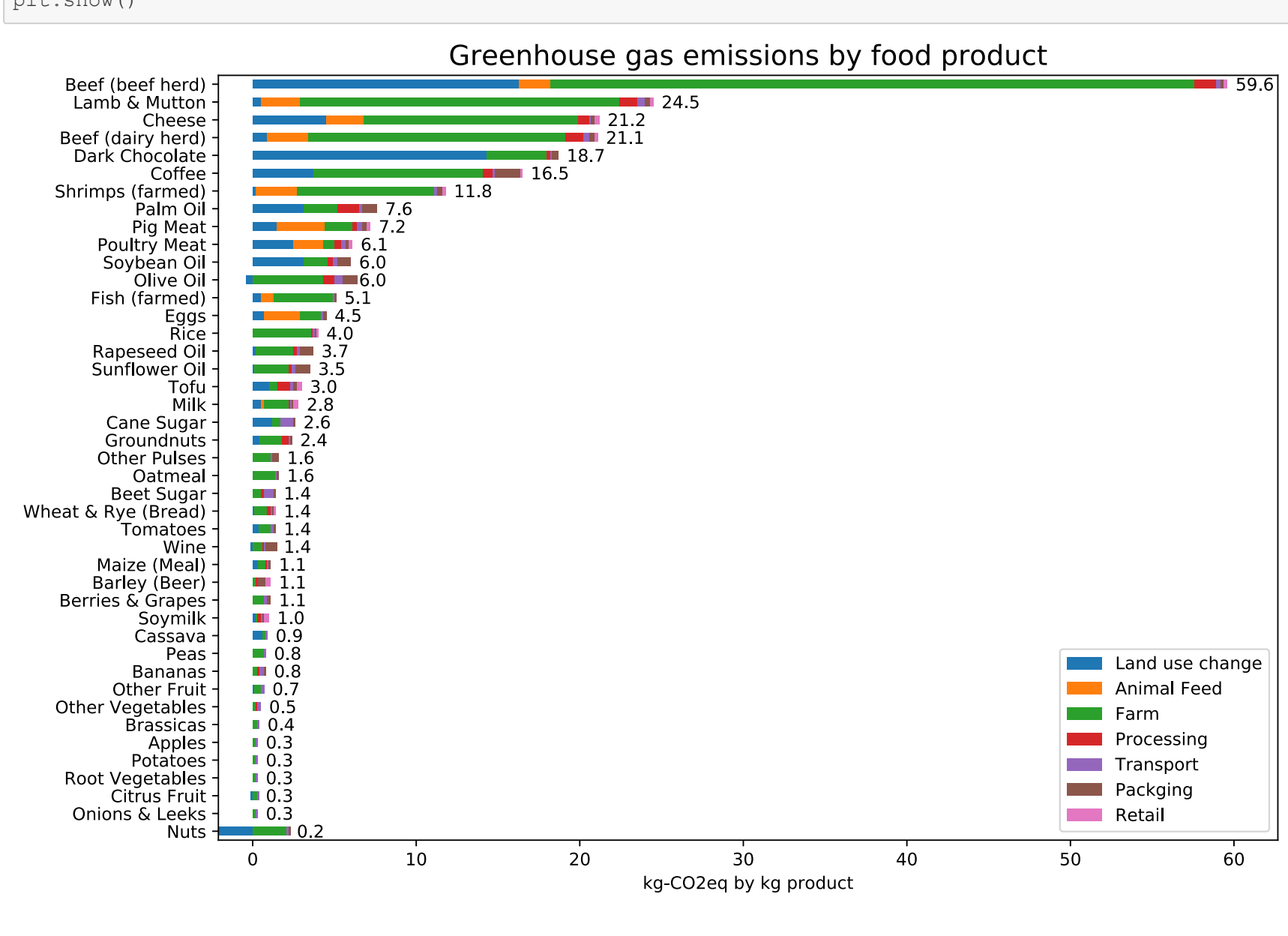
Out[1]:

	Food product	Land use change	Animal Feed	Farm	Processing	Transport	Packging	Retail	Total
0	Wheat & Rye (Bread)	0.1	0.0	0.8	0.2	0.1	0.1	0.1	1.4
1	Maize (Meal)	0.3	0.0	0.5	0.1	0.1	0.1	0.0	1.1
2	Barley (Beer)	0.0	0.0	0.2	0.1	0.0	0.5	0.3	1.1
3	Oatmeal	0.0	0.0	1.4	0.0	0.1	0.1	0.0	1.6
4	Rice	0.0	0.0	3.6	0.1	0.1	0.1	0.1	4.0

Visualizing greenhouse gas emissions by food product

In [3]:
```
# plot greenhouse gas emissions per food product

%config InlineBackend.figure_format = 'svg'
srtid = df_GHG['Total'].sort_values().index
plt.rcdefaults()
ax = df_GHG.loc[srtid, 'Land use change':'Retail'].plot(kind='barh', stacked=True, figsize=(11,8))
ax.set_yticklabels(df_GHG['Food product'][srtid])
ax.set_xlabel('kg-CO2eq by kg product')
ax.set_title('Greenhouse gas emissions by food product', fontsize=16)
for i, v in enumerate(df_GHG['Total'][srtid]):
    if i == 0:
        ax.text(v + 2.5, i - 0.35, str(v))
    else:
        ax.text(v + 0.5, i - 0.35, str(v))
plt.show()
```



Note that in some cases greenhouse gas emissions for land use change are negative (blue bars). This is because the land use for the given product results in lower emissions than what was grown in this land previously (e.g. nut trees store more carbon than crops).

We can see that meat products tend to have much larger associated greenhouse gas emissions than plant products, and food transport (purple bars) somewhat surprisingly have a relatively low overall impact throughout. Hence buying local food might not impact your carbon footprint as much as you might have thought. Read the article on ourworldindata.org or the original paper for a more thorough discussion of the findings.

Acquiring recipes

To get the recipes I had to do a little more work. I was again in luck though, because other developers had already published tools for scraping recipes from recipe-websites that were freely available. By virtue of having published an associated dataset on kaggle that I could use as a starting point for exploratory analyses, I took the code from HugoDarwood, to parse recipe information. And used recipe scrapers, a package that is specifically designed to scrape recipes from a variety of recipe websites, to obtain all recipe links from www.epicurious.com. In total, I scraped approximately 36000 recipes.

The two scripts for scraping recipe links and the information associated with a given recipe link are given below. Note that web-scraping can take a long time and it would make sense to run either of these scripts over night.

Scraping recipe links from www.epicurious.com

In [ ]:
```
from recipe_scrapers import scrape_me

import requests
import re
import pickle
import time


# URL of epicurious search for newest recipes:
initial_search_url = r"https://www.epicurious.com/search/?content=recipe&sort=newest"

# After the first page the url also includes the page number as follows:
# https://www.epicurious.com/search?content=recipe&page=2&sort=newest


# scrape search url and get HTML text
page = requests.get(initial_search_url)
html_text = page.content.decode('utf-8')


# find recipe urls and collect unique recipe links in list
# Example: href="/recipes/food/views/spring-chicken-dinner-salad"
re_rec = r"\/recipes\/food\/views\/(\w+|\-)+"
recipe_links = list(set([x.group() for x in re.finditer(re_rec, html_text)]))


# Go through additional recipes by increasing the page number in the urlimport
start_time = time.time()
pagenum = 2
while True:

    # progress
    if pagenum % 10 == 0:
        print("Page #", pagenum, "Number of recipes scraped = ", len(recipe_links))

    # get next recipe page in HTML
    search_url = r"https://www.epicurious.com/search?content=recipe&page={}&sort=newest".format(pagenum
)
    page = requests.get(search_url)

    # stop looking when max page number is reached
    if page:
        html_text = page.content.decode('utf-8')
        pagenum += 1

        # collect recipe links and append to list
        more_links = list(set([x.group() for x in re.finditer(re_rec, html_text)]))
        recipe_links += more_links
    else:
        print("Reached bottom of page")
        break
print("--- %s seconds ---" % (time.time() - start_time))


# Make sure recipe links are truly unique (should already be)
recipe_links = list(set(recipe_links))


# Save recipe links to txt file
with open('epi_recipe_links', 'wb') as io:
    pickle.dump(recipe_links, io)
```

Scraping recipes

In [ ]:
```
import pickle
import json
import time

# from HugoDarwood, but slightly modified to include servings
# (https://github.com/sbuergers/sustainable-recipe-recommender/blob/master/recipes.py)
from recipes import EP_Recipe


# Load recipe links (from scrape_epicurious_recipe_links.py)
with open('epi_recipe_links', 'rb') as io:
    recipe_links = pickle.load(io)
ep_urls = ["https://www.epicurious.com" + i for i in recipe_links]

print("Scraping recipes from epicurious.....")
start_time = time.time()

# Retrieve recipes in batches and save periodically
output = []
for i, url in enumerate(ep_urls):

    # Convert recipe links to recipe objects (does 16 recipes in ~7-11s)
    output.append(EP_Recipe(url))

    # Convert list of EP_Recipe objects to list of dictionaries
    ar = []
    for iout in output:
        ar.append(iout.__dict__)

    # Dump to json
    if (i % 500 == 0) or (i == len(ep_urls)-1):
        print('Saving recipes....', i, 'out of', len(ep_urls))
        with open('epi_recipes_detailed', 'a') as io:
            json.dump(ar, io)
        output = []

print("--- %s seconds ---" % (time.time() - start_time))
```

We end up with a .json file that has the following structure for each item (I cropped the actual output to be more readable).

```
{'url': 'scallion-pancakes-with-chili-ginger-dipping-sauce',
 'title': 'Scallion Pancakes With Chili-Ginger Dipping Sauce ',
 'ingredients': ['1 (½") piece ginger, peeled,  thinly sliced',
  '2 Tbsp. low-sodium soy sauce',
  ...,
  '4 Tbsp. vegetable oil'],
 'directions': ['Whisk ginger, soy sauce, vinegar, ...',
  'Whisk flour, cornstarch, salt, and sugar in a ...',
  ...,
  'Cut each pancake into wedges if desired and ...'],
 'categories': ['Bon Appétit',
  'Vegetable',
  ...,
  'Pan-Fry'],
 'date': '2020-05-18T13:50:11.682Z',
 'desc': 'These pancakes get their light texture from a batter made with club soda. Pressing har
d on them when frying makes them crisp. \xa0',
 'rating': 3.125,
 'calories': 330.0,
 'sodium': 462.0,
 'fat': 17.0,
 'protein': 5.0,
 'servings': ['4', 'servings']}
```

</id>

Extracting ingredients and their quantities from recipe texts

Aug 2020

This is great! But how are we going to map the ingredients in the recipes to the food categories in the greenhouse gas emissions dataset? What we need is a list of ingredients for all recipes - then we can manually assign a food category label to each ingredient label and create a look-up table from which we can always check the total greenhouse gas emissions of a recipe given the ingredient name and its associated quantity. Obtaining this list of ingredient names and quantities, however, turns out to be somewhat challenging.

Notice that 'ingredients' is a list of strings that contains information about each ingredient. For example, *1 (½") piece ginger, peeled, thinly sliced*, includes the quantity (1 (½")), the ingredient name (ginger), and additional instructions (peeled, thinly sliced). What we want instead is a table like this:

Name	Qnty	Unit
ginger	0.5	Inch

You can probably already see that it is not clear how to reliably extract this information automatically using regular expressions and a set of rules. There can be multiple numbers in a given ingredient line and they do not necessarily need to be combined - here 1 piece and 0.5" are two equivalent descriptions. There can be multiple units, too, or no units at all, and units might not be standardized (a piece of ginger is different from a piece of chicken). There can even be several names that could potentially be the ingredient of interest.

To solve this we turn to machine learning, and luckily for us, once again, there is a an online resource we can use to get us started.

The New York Times ingredients dataset and Conditional Random Fields

The New York Times maintains a recipe database that includes a table similar to what we were looking for. Separate columns are devoted to ingredient name, quantity and unit. Before 2015 data was added to the database by human employees, yielding thousands of rows of structured data. Then, in 2015, Erica Greene, a data scientist at the New York Times made use of this structured, labeled data and trained a machine learning model that takes ingredient text snippets like our example as input:

1 (½") piece ginger, peeled, thinly sliced

and tags different parts of this text according to pre-specified labels (name, quantity, unit, comment, other). The author also wrote a great article about it. The model is a linear chain conditional random field, which can be seen as an extension of logistic regression and is popular in part of speech tagging tasks.

The formula of the model is given as

$$p(y|x) \propto \prod_{t=1}^{T} exp \sum_{k=1}^{K} w_k f_k(y_t, y_{t-1}, x)$$

where y is a sequence of tags, x is a sequence of words, T is the number of words in x, and $\sum_{k=1}^{K} w_k f_k(y_t, y_{t-1}, x)$ denotes the weighted sum of K feature functions. Each feature function can be seen as a conditional statement that returns 1 if some condition is met and 0 otherwise. This includes all combinations of word/label pairs (x_t and y_t), and label/label pairs (y_t and y_{t-1}), and a few custom functions (also see the original article for more details).

$$f_1(y_t, y_{t-1}, x) = \begin{cases} 1 & \text{if } x_t \; is \; flour \cap y_t \; is \; NAME \\ 0 & \text{otherwise} \end{cases}$$

$$f_2(y_t, y_{t-1}, x) = \begin{cases} 1 & \text{if } x_t \; is \; capitalized \cap y_t \; is \; NAME \\ 0 & \text{otherwise} \end{cases}$$

$$f_3(y_t, y_{t-1}, x) = \begin{cases} 1 & \text{if } y_t \; is \; QUANTITY \cap y_{t-1} \; is \; UNIT \\ 0 & \text{otherwise} \end{cases}$$

An iterative optimization algorithm finds the optimal weights.

To actually train this model I used the New York Times dataset with 130000 labeled ingredient phrases, with "resuccitated" code written by Michael Lynch in 2018, which comes in a Linux docker container using python 2.7 and CRF++. For the NYT dataset the model reached 48.73% sentence level accuracy from ~26000 sentences and 73.74% word level accuracy from ~200000 words. While this is a good start, in the future I want to implement my own version in python 3 and create a test dataset to assess model performance on the epicurious data.

Assign greenhouse gas emission estimates to recipes

Aug 2020

After some cumbersome cleaning of my recipe dataset - creating a conversion table of unit tags, a dictionary for quantities, and reducing the number of ingredient names from 30k+ to around 7000 (e.g. by disregarding plurals, different spellings or typos), I was finally ready to go through the ingredient list manually and assign food category labels from the greenhouse gas emission dataset. As with many preprocessing scripts this one is a bit messy and includes a number of steps where I exported and processed the data outside of python by hand. Useful tools were regular expressions and edit distance (or Levenshtein distance - the number of insertions, deletions or alterations needed to convert a string into another string). You can have a look at the full script here.

To illustrate the tediousness of the preprocessing, consider the following conversion table for different units. My struggle might be somewhat evident in the comments. Some units simply weren't precise enough, but I tried to assign the most likely values given what ingredients were most often associated with those units.

In [41]:
```
## Create a look-up table for ingredient amounts
## from wikipedia: https://en.wikipedia.org/wiki/Cooking_weights_and_measures
## Liquid units in milliliters

units_ml = {'drop':0.051,
            'smidgen':0.116,
            'pinch':0.231,
            'dash':0.462,
            'saltspoon':0.924,
            'coffeespoon':1.848,
            'fluid dram':3.697,
            'teaspoon':4.93,
            'dessertspoon':9.86,
            'tablespoon':14.79,
            'ounce':29.57,
            'wineglass':59.15,
            'teacup':118.29,
            'cup':236.59,
            'pint':473.18,
            'quart':946.35,
            'pottle':1892.71,
            'gallon':3785.41,
            'piece':118.29, # this is extremely variable (e.g. ginger, fish, pig, ...)
            'clove':4.93, # A clove of garlic should be around 1 tsp
            'envelope':1.25*29.57, # for yeast
            'pound':16*29.57, # 16 ounces
            'bunch':1.5*29.57, # 1-2 ounces
            'gram':1, # approximately correct, depending on material
            'package':19*29.57, # package of tofu... there is great variability though
            'head':500, # at least a pound for cabbage, cauliflower etc.
            'slice':0.8*29.57, # for a slice of cheese
            'sprig':1.848,
            'can':12*29.57, # for a small can
            'stick':4*29.57, # for a butter stick
            'strip':29.57,
            'stalk':59.15, # for a stalk of celery (could also be lemongrass)
            'cube':4.93, # should be 1 teaspoon (always sugar)
            'fillet':100, # 100 grams is roughly 1 fillet
            'handful':118.29, # by definition it's half a cup
            'fistfull':59.15, # by definition it's half a handful
            'bag':11*29.57, # 10-12 ounces it seems, though probably not always
            'loaf':6*236.59, # approximately 4-8 cups
            'bulb':8*29.57, # for a bulb of fennel
            'bottle':1.25*473.18, # could be beer or wine
            'ear':(3/4)*236.59, # for an ear of corn
            'ball':236.59, # a ball of mozzarella is a cup
            'batch':473.18, # pretty unclear, milk, eggs, fish, fruit...
            'sheet':14*29.57, # sheet of pastry dough or cheese
            'dozen':473.18, # used for clams - roughly one pound
            'liter':1000,
            'box':473.18, # e.g. box of chocolate or milk
            'packet':59.15, # can be many sizes, usually quite small (1/4 ounce), but can also be a pac
ket of dumpling wrappers
            'chunk':1.1*14.79, # used for ginger pretty much
            'rack':3.5*16*29.57, # a rack of ribs... 3-4 pounds
            'jar':236.59, # pretty variable...
            'stem':14.79, # a stem of thyme
            'part':118.29, # bread, fruit and sugar... no clear amount, few entries though
            'branch':14.79,
            'inch':1.1*14.79, # used only for ginger
            'wedge':6*29.57, # a wedge of cheese is 4-8 ounces
            'link':29.57, # a link of sausage is 1 ounce
            'square':29.57, # a square of chocolate - 1 ounce
            'knob':118.29, # same as piece
            'scoop':59.15, # as much as a wineglass by definition
            'fifth':757, # a fifth of a gallon for liquor - wow
            'twist':29.57, # refers to lemon or orange peel
            'pair':172*2 # because in this case it's chicken breast !!!
            }
```

Of course, my manual conversion from ingredient labels to food categories is faulty in some cases, but I decided that it would be less detrimental to assign some items to somewhat incorrect food groups, rather than not assigning a value at all (which would essentially be taken as an emission score of 0). After assigning all "basic" ingredients to food groups, I used the recipe dataset itself to assign greenhouse gas emission estimates to more complex ingredients that really consist of several basic ingredients (e.g. mayonnaise). In the end I was able to assign greenhouse gas emission estimates to 333728 out of 354192 total ingredients (94%). Not bad.

In [112]:
```
## load recipe data

df_rec = pd.read_csv(r'D:\data science\nutrition\data\recipes_sql.csv', index_col=0)

df_rec.head(2)
```

Out[112]:

recipesID	title	ingredients	directions	categories	date	desc	rating	calories	sodium
0	Scallion Pancakes With Chili-Ginger Dipping Sa...	1 (½") piece ginger, peeled, thinly sliced;2 ...	whisk ginger, soy sauce, vinegar, chili oil, a...	bon appétit;vegetable;side;green onion/scallio...	2020-05-18T13:50:11.682Z	these pancakes get their light texture from a ...	2.5	330.0	462.0
1	Triple-Threat Onion Galette	1 tbsp. sugar;1½ tsp. diamond crystal or 1 tsp...	whisk sugar, salt, and 1½ cups flour in a medi...	bon appétit;bake;dinner;lunch;brunch;vegetaria...	2020-05-15T20:53:41.255Z	the key to this flaky galette crust is to move...	5.0	3708.0	4504.0

Visualize recipe greenhouse gas emissions

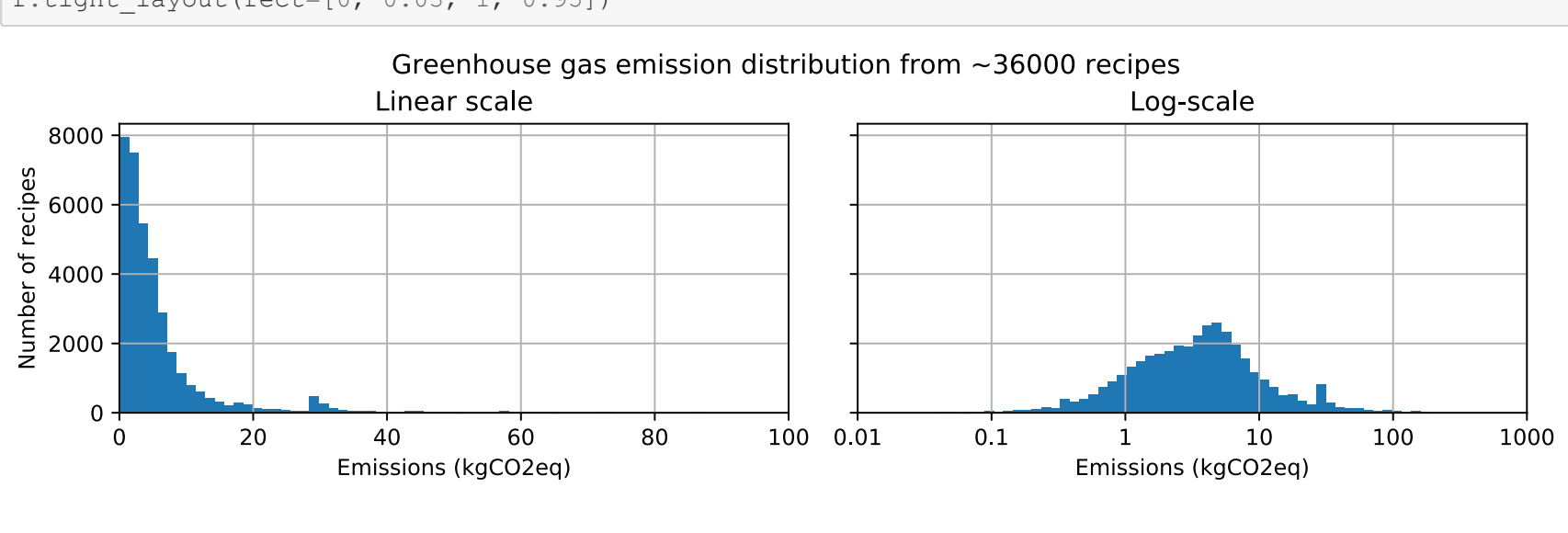
In [113]:
```
## Histogram of greenhouse gas emissions over all recipes (after preprocessing)

f, (ax1, ax2) = plt.subplots(1, 2, sharey=True, figsize=(10,3))

ax1.hist(df_rec['ghg'].dropna(), bins=80)
ax1.set_title('Linear scale')
ax1.set_ylabel('Number of recipes')
ax1.set_xlabel('Emissions (kgCO2eq)')
ax1.grid()

ax2.hist(df_rec['ghg_log10'].dropna(), bins=80)
ax2.set_xlim(-2, 3)
ax2.set_xlabel('Emissions (kgCO2eq)')
ax2.set_xticklabels([0.01, 0.1, 1, 10, 100, 1000])
ax2.set_title('Log-scale')
ax2.grid()

f.suptitle('Greenhouse gas emission distribution from ~36000 recipes')
f.tight_layout(rect=[0, 0.03, 1, 0.95])
```



Let's explore some example recipes in the first, second and third tercile of the greenhouse gas emission distribution

In [114]:
```
# log-greenhouse gas emission terciles:
cutoff1 = sorted(df_rec['ghg'])[12000]
cutoff2 = sorted(df_rec['ghg'])[24000]
print('First cutoff:', cutoff1, 'kg CO2 equivalent')
print('Second cutoff:', cutoff2, 'kg CO2 equivalent')
```

```
First cutoff: 2.108655626956509 kg CO2 equivalent
Second cutoff: 5.17105997873272 kg CO2 equivalent
```

In [117]:
```
# create group vector (0 = low, 1 = medium, 2 = high)
GHG_group = np.zeros(df_rec.shape[0])
GHG_group[df_rec['ghg'].between(cutoff1, cutoff2)] = 1
GHG_group[df_rec['ghg'].between(cutoff2, 99999)] = 2
df_rec['GHG group'] = GHG_group

## Show 10 recipes for each subgroup
print(' ')
print('Sustainable:')
print(df_rec['title'][df_rec['GHG group'] == 0][0::1000])
print(' ')
print('Not sustainable:')
print(df_rec['title'][df_rec['GHG group'] == 1][0::1000])
print(' ')
print('Absolutely not sustainable:')
print(df_rec['title'][df_rec['GHG group'] == 2][0::1000])
```

```
Sustainable:
recipesID
0        Scallion Pancakes With Chili-Ginger Dipping Sa...
3186                         Overeater's Tonic (or Green ...
6142     Guacamole with Bacon, Grilled Ramps (or Green ...
9155                                  Homemade Ginger Beer
12478                                 Rick's Basic Crostini
15687                                   Passion Fruit Syrup
18194                                    Lemon Vodka Gimlet
20001    Baked Salmon with Tarragon and Fennel-Seed But...
23334    Butternut Squash and Carrot Puree with Maple S...
26598                                                 Aioli
29782                          Stuffed Portabella Mushrooms
32934    Caramelized Pineapple with Brown Sugar-Ginger ...
Name: title, dtype: object

Not sustainable:
recipesID
3                          Sour Cream and Onion Biscuits
3043                           Wild Mushroom Noodle Soup
6036                                  Miso-Yaki Sea Bass
8996     Grilled Salmon with Black Beans and Piment d'E...
11986                         Lemon-Ginger Frozen Yogurt
15041                          Herbed Goat Cheese Toasts
18554                                       Basic Eggnog
21751                           Honey Lavender Ice Cream
24652                                       Turkey Stock
27629      Roasted Quail with Red Grapes and Pearl Onions
30452                              Espresso Pastry Cream
33400                          Ranch-Style Poquito Beans
Name: title, dtype: object

Absolutely not sustainable:
recipesID
2                               Cinnamon-Date Sticky Buns
2744              Crunchy Veg Bowl with Warm Peanut Sauce
5783                                    Barbecued Chicken
8866     Linguine with Baby Heirloom Tomatoes and Ancho...
11632    Cheese-Filled Risotto Croquettes with Tomato S...
14329           Prune, Apple, and Chestnut Bread Pudding
17154                                   Calgary Hot Wings
21139    Seared Scallops with Pea Puree, Crispy Bacon, ...
23907          Beef Rolls with Ham and Vegetables Stuffing
26781       Beef Chili with Chipotle Chilies and Cilantro
29228                    Chicken and Mushroom Quesadillas
32600               Crispy Masa Boats with Pork Picadillo
35480             Beet Flowers and Beet Greens Vinaigrette
Name: title, dtype: object
```

The distribution of the example recipes makes some sense. Meat and dairy products seem to increase from low toward high carbon emissions. Just be aware that not all recipes are correctly placed - e.g. *Beet flowers and beet greens vinaigrette* is unlikely to rank in the top 1/3 most polluting recipes. See the greenhouse gas estimates as a guide, use common sense and refer to the bar chart of greenhouse gas emissions for essential food products to get a feel for when the algorithms' estimate might be off. I plan to include a confidence score for a recipe's sustainability estimate in the future.

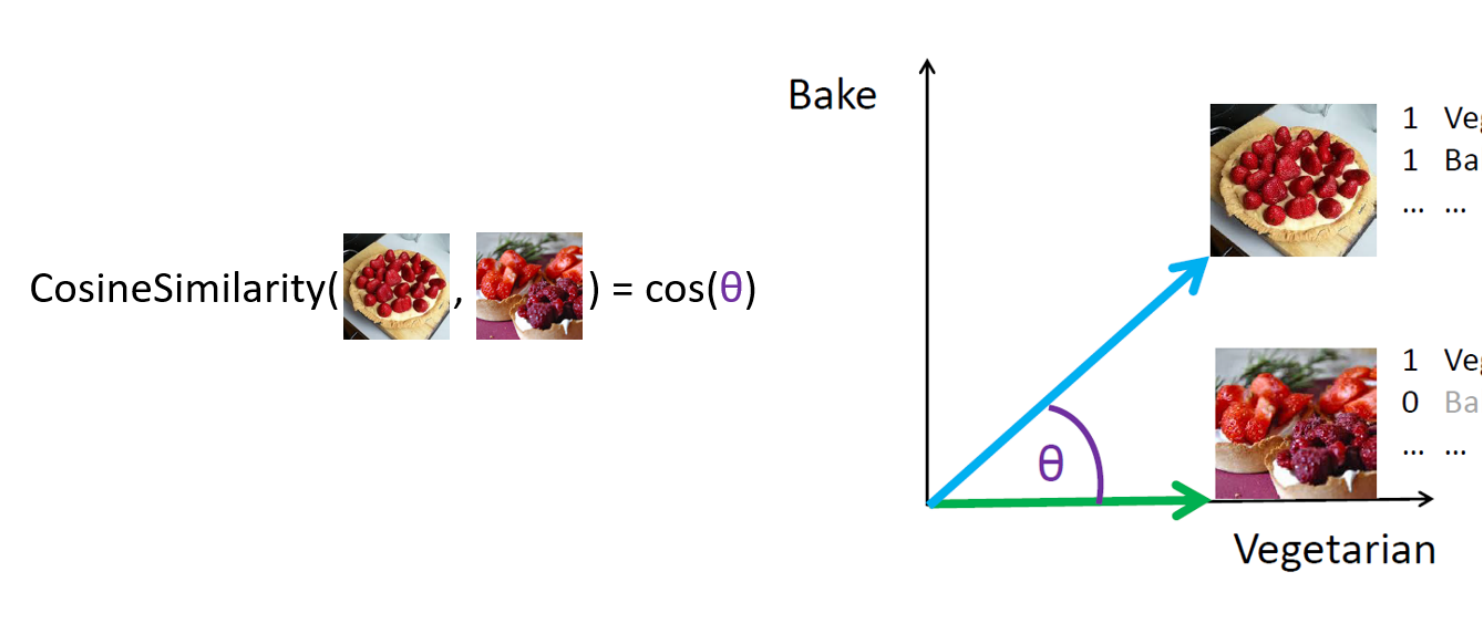
Building a recommender based on category labels

Aug 2020

Making recommendations to users is an important feature of online stores (e.g. Ebay), streaming services (e.g. Netflix) and indeed recipe websites (e.g. Yummly). Given the ubiquitousness of personal recommenders it is not surprising that there has been a lot of research on how to improve and tailor them for specific use cases.

For my minimum viable product we simply want to recommend recipes similar to a reference recipe that the user gives as an input. For instance, if I want to bake a strawberry pie I might feed the strawberry pie recipe I know and love into the algorithm (in the form of a recipe URL from epicurious), and in turn receive suggestions of similar products, likely also strawberry pies. To do this we need a measure that assesses the similarity between different recipes. A common choice is cosine similarity, which measures the angle between a pair of recipes in an N dimensional feature space in which each dimension represents a relevant aspect for assessing recipe similarity.

Let's make this more concrete. Consider the strawberry pie in the figure below to be our input recipe, and the little strawberry or raspberry tarts to be a comparison recipe. The strawberry pie has the labels "Vegetarian" and "Bake", whereas the fruit tarts have the label "Vegetarian", but do not have the label "Bake". We associate having a given label with a 1 and 0 otherwise. Hence our input recipe will be associated with the vector (1, 1), whereas the comparison recipe has the vector (1, 0). The cosine similarity is simply the cosine of the angle between these two vectors (which happens to be 0.71 in this case). Cosine similarity ranges between 0 (cosine of 90 degrees) and 1 (cosine of 0 degrees), indicating orthogonal and identical items respectively. Of course, in reality there are many more categories than just "Vegetarian" or "Bake", but the underlying idea and math are exactly the same.



The categories for the different dimensions were taken from the "categories" column of the epicurious dataset, with just over 700 categories in total (including descriptive labels such as "Bake", "Gluten-free", "under 20 minutes", as well as a subset of ingredient labels. Since a given recipe can already have a rather low carbon footprint, it actually makes sense to disregard greenhouse gas emissions when making recommendations. This way the user can feel good about using a recipe that has low carbon emissions and compare where it ranks in the distribution of similar recipes. Conversely, when the input recipe has high carbon emissions, most suggestions will have lower emissions anyway. Try it out!

While this content-based similarity approach works well for recommending similar recipes, it can be useful to also be able to suggest recipes that are not exactly similar in terms of category labels, but which the user would probably also enjoy. For this, collaborative filtering techniques can be employed, which make recommendations based on the similarity of a given user's liked recipes to the liked recipes of other users. For example, if I liked recipes A, B and C, and another user liked recipes A, B, C, and D, it is not much of a stretch to assume that I might also like recipe D. Including this functionality is currently a work in progress.

Building the website

Aug 2020

Building the website was more cumbersome than I anticipated, but I managed to cook up a reasonably simple web-app with a search functionality for reference recipes using a Flask backend, Jinja2 for passing variables from python/Flask and handling repetitive HTML, some CSS, as well as a little bit of JavaScript for passing Altair plots and creating visuals for ratings, similarity and sustainability scores. The code is available on my github, and the website can be visited at this Heroku domain.

Initially, I wanted to create a PostgreSQL database for the recipe dataset, but discovered that the number of rows (i.e. recipes) is too large for the free tier on Heroku. So instead, I added the data as .csv files to object storage. However, as it turns out (quite logically), this is not really a viable deployment option, because Heroku's free RAM is only ~500 MB, and loading the full .csv table(s) for different dynos or users will quickly make it overflow. To still get away with free storage, I then decided to use AWS RDS directly (not via Heroku, which uses the same service under the hood), which gives 1 year free usage with up to 20GB storage - more than enough space for my purposes.

For anyone who wants to attempt the same be aware that 1.) when connecting to your DB instance on AWS RDS, be sure to not enter the name you gave it, but simply "postgres" (see this article), and 2.) you need to ensure your request does not get blocked by a firewall. For this you need to change your security group settings.

I ended up creating my SQL schemas and tables using both pgadmin4 (a GUI) and psycopg2 (programmatically). For suggesting similar recipes I created three tables: 1.) A recipes table where each row is one recipe, 2.) a content based similarity table, where each row is one recipe and columns denote progressively less similar recipes, with similarity scores, and 3.) a content based similarity recipeID table, where each cell contains the recipe ID.

1.) Recipes table

recipesID	title	ingredients	rating	emissions	...
0	'Burger'	'1 large ...'	4.5	3.2	...
1	'Fries'	'2 pnds ...'	3.8	0.8	...
...	...	...	...	...	...

2.) Content based similarity table

recipesID	0	1	...	199
0	1	0.49	...	0.28
1	1	0.81	...	0.55
...	...	...	...	...

3.) Content based similarity recipeID table

recipesID	0	1	...	199
0	0	1233	...	3924
1	1	4833	...	23789
...	...	...	...	...

Here is some sample code for retrieving similar recipes for a given reference recipe.

In [ ]:
```
# for loading environment variables
import os

# needed to load environment variables from .env file
from dotenv import load_dotenv

# for connecting to postgres database
import psycopg2 as ps
from psycopg2 import sql


# need to load variables from os.environ.get (.env file)
load_dotenv()


# create connection and cursor
conn = ps.connect(host=os.environ.get('AWS_POSTGRES_ADDRESS'),
                  database=os.environ.get('AWS_POSTGRES_DBNAME'),
                  user=os.environ.get('AWS_POSTGRES_USERNAME'),
                  password=os.environ.get('AWS_POSTGRES_PASSWORD'),
                  port=os.environ.get('AWS_POSTGRES_PORT'))

cur = conn.cursor()


# How do I now use the sql output to recommend recipes?
# For now do exact searches, later explore partial matching in sql
search_term = 'mango-toast-with-hazelnut-pepita-butter'


# Select recipe IDs of 200 most similar recipes to reference (search_term)
cur.execute(sql.SQL("""
            SELECT * FROM public.content_similarity200_ids AS csids
            WHERE "recipeID" = (
                SELECT "recipesID" FROM public.recipes
                WHERE url = %s)
            """).format(), [search_term])
CS_ids = cur.fetchall()[0][1::]
CS_ids = tuple([abs(CSid) for CSid in CS_ids])


# Also select the actual similarity scores
cur.execute(sql.SQL("""
            SELECT * FROM public.content_similarity200
            WHERE "recipeID" = (
                SELECT "recipesID" FROM public.recipes
                WHERE url = %s)
            """).format(), [search_term])
CS = cur.fetchall()[0][1::]
CS = tuple([float(abs(s)) for s in CS])


# Finally, select similar recipes themselves
# Get only those columns I actually use to speed things up
col_sel = [
          'recipesID', 'title', 'ingredients', 'rating', 'calories', 'sodium',
          'fat', 'protein', 'ghg', 'prop_ing', 'ghg_log10', 'url', 'servings',
          'index'
           ]

# I am aware there is a discrepancy between the column names in SQL
# and python, which is rather ugly. It is on my todo list to fix
# in the future.
cur.execute(sql.SQL("""
            SELECT "recipesID", "title", "ingredients",
                   "rating", "calories", "sodium", "fat",
                   "protein", "emissions", "prop_ingredients",
                   "emissions_log10", "url", "servings", "recipe_rawid"
            FROM public.recipes
            WHERE "recipesID" IN %s
            """).format(), [CS_ids])
recipes_sql = cur.fetchall()


# Obtain a dataframe for further processing
results = pd.DataFrame(recipes_sql, columns=col_sel)
results['similarity'] = CS


cur.close()
```

For convenience and future extendability I created a class for dealing with DB connection and querries (see the full class here). I also quickly cooked up a very simple free search functionality, which uses Postgres' LIKE operator and edit distance as a measure of similarity. There are many flaws with this approach, but for simple key words it gives mostly reasonable results if the keyword is embedded within the url string of recipes (e.g. "chicken" is embedded in "cashew-chicken"). It really does not work very well when the keyword is not embedded in the url at the moment and the current output should be seen as a stand-in (giving any output) for a future, better implementation. I might consider using ElasticSearch, although using PostgreSQL's free-text-search functionality also sounds promising since I would not have to manage two data stores at once.

@_dbsrr_query is a dcorator, which tries to reconnect to the database if the connection is lost while performing the query.

In [ ]:

```python
# Postgres connection class used for all interactions with the postgres AWS DB
class postgresConnection():

    def __init__(self):
        self.connect()

    def connect(self):
        '''
        DESCRIPTION:
            Create connection to AWS RDS postgres DB and cursor
        '''
        self.conn = ps.connect(
                        host=os.environ.get('AWS_POSTGRES_ADDRESS'),
                        database=os.environ.get('AWS_POSTGRES_DBNAME'),
                        user=os.environ.get('AWS_POSTGRES_USERNAME'),
                        password=os.environ.get('AWS_POSTGRES_PASSWORD'),
                        port=os.environ.get('AWS_POSTGRES_PORT'))
        self.cur = self.conn.cursor()

    def _dbsrr_query(func):
        """
        DECORATOR: Basically a try except for functions that query the postgres
            DB. When the connection fails, it tries to reconnect automatically
        """
        def func_wrapper(self, *args, **kwargs):
            try:
                return func(self, *args, **kwargs)
            except ps.OperationalError:
                return self.connect()
        return func_wrapper

    @_dbsrr_query
    def fuzzy_search(self, search_term, search_column="url", N=160):
        """
        DESCRIPTION:
            Searches in recipes table column url for strings that include the
            search_term. If none do, returns the top N results ordered
            by edit distance in ascending order.
        INPUT:
            cur: psycopg2 cursor object
            search_term (str): String to look for in search_column
            search_column (str): Column to search (default="url")
            N (int): Max number of results to return
        OUTPUT:
            fuzzyMatches (list): DB output (list of lists - rows x columns)
        """
        # Most similar urls by edit distance that actually contain the
        # search_term
        self.cur.execute(sql.SQL(
                    """
                    SELECT "recipesID", "title", "url", "perc_rating",
                        "perc_sustainability", "review_count", "image_url",
                        "emissions", "prop_ingredients",
                        LEVENSHTEIN({}, %s) AS "edit_dist"
                    FROM public.recipes
                    WHERE {} LIKE %s
                    ORDER BY "edit_dist" ASC
                    LIMIT %s
                    """).format(sql.Identifier(search_column),
                                sql.Identifier(search_column)),
                    [search_term, '%'+search_term+'%', N])
        fuzzyMatches = self.cur.fetchall()

        # If no results contain the search_term
        if not fuzzyMatches:
            self.cur.execute(sql.SQL(
                        """
                        SELECT "recipesID", "title", "url", "perc_rating",
                            "perc_sustainability", "review_count", "image_url",
                            "emissions", "prop_ingredients",
                            LEVENSHTEIN({}, %s) AS "edit_dist"
                        FROM public.recipes
                        ORDER BY "edit_dist" ASC
                        LIMIT %s
                        """).format(sql.Identifier(search_column)),
                        [search_term, N])
            fuzzyMatches = self.cur.fetchall()
        return fuzzyMatches
```

Work in progress

Aug 2020
last edited: Oct 07 2020

For an updated tally of issues and enhancements see the github repo.

i. Improve free search.

ii. Create a test set of ingredient tags by hand, to assess CRF performance. Then tweak the model for improved predictions.

iii. Users can log in and add recipes to their cookbook (similar to liking a recipe).

iv. When #3 works, I can include user-user based recommendations using for instance SVD++.

v. Add a confidence score to sustainability ratings (e.g. based on how many ingredients were assigned, or how sure the CRF model was when assigning labels to parts of speech)

vi. Incorporate unit tests, integration tests etc. - ultimately aim for continous integration.

vii. Make sure website is safe - no sql injection or cross site scripting, no embedding in unknown websites, force ssl to prevent phishing.

Scraping recipe reviews for recommendations

Aug 2020

Scraping recipe reviews was more cumbersome than other recipe information (ingredients, servings, etc.). Actually, getting up to 25 reviews was easy using the recipe scrapers toolbox and beautiful soup. However, beyond 25 recipe reviews the website has a "view more reviews" button that the user has to click in order to load an additional 25 reviews (see for yourself if you like). There is no change in the url, and there is no information about those additional recipes in the source code up until that button is clicked.

Therefore, I decided to use Selenium, which is used to simulate the behaviour of users and debug websites, to actually open the website in a browser window, scroll down and click the button until it no longer exists. The solution works pretty well, but takes a long time. Hence, I first scraped all reviews that are visible on the initial recipe page, and only applied the selenium approach when there were 25 total reviews (hence there could be more). In addition, in some cases the website did not load fast enough leading to my code timing out and requiring a re-scrape for those occasions.

In [ ]:

```python
# Scrape recipe reviews (25 or less)

# Package for scraping recipes from many popular websites, for details see
# https://github.com/sbuergers/recipe-scrapers/blob/master/recipe_scrapers/epicurious.py
from recipe_scrapers import scrape_me

# Data management
import pandas as pd
import json
import pickle

# Check execution time
import time


# Load recipe links (from scrape_epicurious_links.py)
with open('epi_recipe_links', 'rb') as io:
    recipe_links = pickle.load(io)

ep_urls = ["https://www.epicurious.com" + i for i in recipe_links]
start_time = time.time()

# Set filename
timestr = time.strftime("%Y%m%d_%H%M%S") # make filename unique for every run
filename = 'epi_reviews' + timestr + '.txt'

# Go through all files that are not already in filename
try:
    with open(filename, 'r') as io:
        old_reviews = json.load(io)
    S = len(old_reviews.keys())
except:
    S = 0
N = len(ep_urls)

review_dict = {}
for i, url in enumerate(ep_urls[S:N]):

    # scrape reviews from recipe page
    scraper = scrape_me(url)
    reviews = scraper.reviews()

    # Add recipe to review dictionary
    webpart = 'https://www.epicurious.com/recipes/food/views/'
    pruned_url = url[len(webpart)::]
    review_dict[pruned_url] = reviews

    # Progress
    if i % 100 == 0:
        print(i, url)

# Code timing
print("--- %s seconds ---" % (time.time() - start_time))


# Save reviews dictionary to json (append every 1000 recipes)
with open(filename, 'w') as io:
    json.dump(review_dict, io)


## eof
```

Scrape additional reviews using Selenium

In [ ]:

```python
# Package for scraping recipes from many popular websites, for details see
# https://github.com/sbuergers/recipe-scrapers/blob/master/recipe_scrapers/epicurious.py
from recipe_scrapers import scrape_me

# Get HTML from website
import requests

# Regular expressions
import re

# Check for files / paths
import os.path
from os import path

# Data management
import pandas as pd
import json
import pickle

# Check execution time
import time

# parsing page (scrape_me wants url, not text of page)
from bs4 import BeautifulSoup as bs

# Get selenium to "press" load more reviews button (there should be an easier
# way to do this, but not sure how)
## From
## https://codereview.stackexchange.com/questions/169227/
## scraping-content-from-a-javascript-enabled-website-with-load-more-button
from selenium import webdriver
from selenium.common.exceptions import NoSuchElementException, StaleElementReferenceException, ElementC
lickInterceptedException
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.common.action_chains import ActionChains
from selenium.webdriver.common.keys import Keys


def get_load_reviews_button(driver):
    """Returns the load more reviews button element if it exists"""
    try:
        return driver.find_element(By.XPATH, '//button[text()="View More Reviews"]')
    except NoSuchElementException:
        return None


def center_page_on_button(driver, button):
    """Gets the load more reviews button into view (so it's clickable) """
    try:
        if button:
            driver.execute_script("arguments[0].scrollIntoView();", button)
            driver.execute_script("window.scrollBy(0, -150);")
    except:
        raise


def click_load_reviews_button(button):
    """Attemps to hover over and click the load more views button """
    try:
        button.click()
        return "button_clicked"
    except StaleElementReferenceException:
        return "no_button"
    except AttributeError:
        return "no_button"
    except ElementClickInterceptedException:
        return "pop_up_interferes"
    except:
        raise


def close_pop_up(driver):
    """Makes selenium 'press' the ESC key to close pop-up window """
    webdriver.ActionChains(driver).send_keys(Keys.ESCAPE).perform()


def get_expanded_reviews_page(driver, fullurl):
    """Expands all recipe reviews of the given epicurious url by 'clicking'
    the view more recipes button until it disappears. Returns html page. """
    ## Connect to Epicurious recipe URL
    driver.get(fullurl)

    # Do we have a load more reviews button?
    button = get_load_reviews_button(driver)

    # If so, attempt to click the Load Reviews Button until it vanishes
    if button:

        # center page on load more reviews button
        center_page_on_button(driver, button)

        # click the button
        status = click_load_reviews_button(button)

        # Keep doing this until the button disappears or we time out with an error
        start_time = time.time()
        run_time = 0
        timeout = 90
        while (button) and (not status == "no_button") and (run_time < timeout):
            if status == "pop_up_interferes":
                close_pop_up(driver)
            button = get_load_reviews_button(driver)
            center_page_on_button(driver, button)
            status = click_load_reviews_button(button)
            run_time = time.time()-start_time

    return driver.page_source


## Since recipe scrapers internally uses requests and only takes url as input,
## rather than rewriting the toolbox to also accept page content, adapt the
## function that gets users reviews and include it here:
def get_reviews(page):
    """Scrapes review texts from epicurious web-pages. Page is the HTML of the
    web-page. result is a dictionary with 'review_text' and 'rating' as keys,
    including a string and integer as values, respectively."""
    fork_rating_re = re.compile('/(\d)_forks.png')
    soup = bs(page, 'html.parser')
    reviews = soup.findAll('', {'class': "most-recent"})
    ratings = [rev.find('img', {'class': "fork-rating"}) for rev in reviews]
    temp = []
    for rating in ratings:
        if 'src' in rating.attrs:
            txt = rating.attrs['src']
        else:
            txt = ''
        rating = fork_rating_re.search(txt)
        rating = rating.group(1) if rating is not None else '0'
        rating = int(rating) if rating != '0' else None
        temp.append(rating)
        ratings = temp
    review_texts = [rev.find('div', {'class': "review-text"}) for rev in reviews]
    reviews = [rev.get_text().strip('/ flag if inappropriate') for rev in review_texts]
    result = [
        {'review_text': review_text, "rating": rating_score}
        for review_text, rating_score in zip(reviews, ratings)
        ]
    return result


# Setup selenium webpage
# Includes adding adblock extension and skipping loading of images
# NOTE: Occasionally restarting the driver speeds the process up tremendously!
def initialize_selenium_session():
    """Initiates a selenium chrome session without loading images and an
    adblock extension."""
    prefs = {'profile.managed_default_content_settings.images': 2}
    chrome_options = webdriver.ChromeOptions()
    chrome_options.add_extension(r'D:\data science\nutrition\misc\AdBlockPlus.crx')
    chrome_options.add_experimental_option('prefs', prefs)
    driver = webdriver.Chrome(options=chrome_options)
    time.sleep(10) # wait a few seconds for chrome to open
    return driver


# recipe-scrapers works beautifully for recipes with less than 25
# reviews. Here we are only looking at recipes with more than 25 reviews,
# because using selenium to click the "load more reviews" button is slow.


# Load recipe links (from scrape_epicurious_recipe_reviews.py)
with open('epi_reviews20200619_232923.txt', 'r') as io:
    reviews = json.load(io)

# Initialize Selenium browser session
driver = initialize_selenium_session()

# Add "hidden" reviews where necessary
start_time = time.time()
faillog = []
reviews_new = {}
for i, url in enumerate(reviews.keys()):

    # Only run over a subset (e.g. already did the first 5000):
    if i < 3000: # change manually!
        continue

    if len(reviews[url]) == 25:

        # Sometimes it simply does't work, retry a few times, otherwise
        # remember where it failed
        num_tries = 0
        no_success = True
        while (num_tries < 5) and (no_success):
            try:
                # Get html text of full page (with all reviews)
                webpart = 'https://www.epicurious.com/recipes/food/views/'
                page = get_expanded_reviews_page(driver, webpart + url)

                # scrape reviews from recipe page
                page_reviews = get_reviews(page)

                # Update review dictionary with additional reviews
                reviews[url] = page_reviews
                no_success = False
            except:
                num_tries += 1
        if num_tries == 5:
            faillog.append([i, url])

        print('Adding new reviews:', i, url, len(reviews[url]))

    # Save periodically
    reviews_new[url] = reviews[url]
    if (i+1) % 200 == 0 | (i==len(reviews)):

        # Saving dictionaries is a bit of a pain if done recurrently,
        # but I can simply load in the previous dictionary and append
        if path.exists('epi_reviews_25plus.txt'):
            with open('epi_reviews_25plus.txt', 'r') as io:
                reviews_old = json.load(io)
            reviews_to_file = {**reviews_old, **reviews_new}
        else:
            reviews_to_file = reviews_new

        # Save reviews dictionary to json
        with open('epi_reviews_25plus.txt', 'w') as io:
            json.dump(reviews_to_file, io)
        reviews_new = {}

        # Write fail-log to file
        with open('epi_reviews_25plus_faillog.txt', 'a') as io:
            for item in faillog:
                io.write('%s\n' % item)
        faillog = []

        print('\n ----- Saving to file ----- \n')

    # As Chrome slows down over time, it makes sense to periodically restart
    # the Selenium session (i.e. close and restart)
    if (i+1) % 1000 == 0:
        driver.quit()
        driver = initialize_selenium_session()

# Code timing
print("--- %s seconds ---" % (time.time() - start_time))


# Tidy up Selenium browser session
driver.quit()


# eof
```

Improved free search

Oct 07 2020

Postgres has some excellent features for designing a good search engine. See for instance this blog post for a more thorough discussion. Basically, we can convert text to a series of lexemes, in which we keep track of the lexemes' positions in the text. The function to do this is called `to_tsvector`.

For example, the input `Butternut Squash with Shallots and Sage` is converted to `'and':5 'butternut':1 'sage':6 'shallots':4 'squash':2 'with':3`.

Now, it's possible to search for a phrase - e.g. `Squash and Sage` - and find a match, even when there are other words in-between (using the `<->` operator). Results where these words are closer together will be preferred. Of course, there are a variety of other ways to search, like forcing to have a word not occur (`!word`), having multiple words occur (`word1 & word2`) or having one of a number of words occur in the results (`word1 | word2`). The function to design a search query is called `to_tsquery`, which has a few variants. We are going to use `websearch_to_tsquery` as it will be lenient with badly formatted input given by a user - i.e. not throw errors all the time.

Another great feature of postgres is that we can actually combine multiple inputs into a single tsvector. On top of that we can assign which of the inputs should be given a larger weight in the search results ranking. I opted for using both the recipe title (e.g. Butternut Squash with Shallots and Sage) and categories columns (e.g. Vegan, gluten-free, beef, ...). With this simple setup the free search functionality already improved a lot.

When no results are found the user is presented with the recipes where the title is closest to the input in terms of edit distance (the numbef of insertions, deletions or substitutions necessary to convert one string to another). This often gives rather poor results, but avoids showing no results at all. It also simply works as a stand-in for what I might opt to do when no results are found in the future.

In [ ]:

```python
@_dbsrr_query
def phrase_search(self, search_column, search_term, N=160):
    """
    DESCRIPTION:
        Searches in table recipes in combined_tsv column using tsquery
        - a tsvector column in DB table recipes combining title and
        categories.
    INPUT:
        cur: psycopg2 cursor object
        search_column (str): Name of table column to search
        search_term (str): Search term
        N (int): Max number of results to return
    OUTPUT:
        matches (list[list]): DB query result
    """
    self.cur.execute(sql.SQL(
        """
        SELECT "recipesID", "title", "url", "perc_rating",
            "perc_sustainability", "review_count", "image_url",
            "emissions", "prop_ingredients",
            ts_rank_cd({search_column}, query) AS rank
        FROM public.recipes,
            websearch_to_tsquery('simple', {search_term}) query
        WHERE query @@ {search_column}
        ORDER BY rank DESC
        LIMIT {N}
        """).format(
            search_column=sql.Identifier(search_column),
            search_term=sql.Literal(search_term),
            N=sql.Literal(N)
        )
    )
    matches = self.cur.fetchall()
    return matches
```

Incorporate tests

Oct 07 2020

Tests that automatically assess whether the app still runs as intended are incredibly useful, but somewhat cumbersome to create. Especially, when not considering to do so from the outset.

Using pytest (and trying to stick only with pytest), I created some simple unit-tests (tests for single functions) for both sql querries and the main application script. There is certainly still room for improvement and more tests need to be written. But having only this simple setup already helped quite a bit as the tests have already failed here and there and pointed me toward problems as I made changes.

Below are the unit tests for sql_queries.py. `@pytest.fixture` is a way of passing certain arguments to test functions that are the same for multiple functions. In addition it can be used to setup the database connection. See here for more information about fixtures. Also note that it is possible to group test functions in a class when the class name begins with "Test".

In [ ]:

```python
"""
Unit tests for sql_queries.py
"""
import pytest
import psycopg2 as ps

# Make sure parent directory is added to search path before
# importing sql_queries!
import os
import sys
currentdir = os.path.dirname(os.path.realpath(__file__))
parentdir = os.path.dirname(currentdir)
sys.path.append(parentdir)

# Now I can import sql_queries
import sql_queries
from dotenv import load_dotenv

load_dotenv('.env')


# I cannot simply define an __init__ method - pytest does
# not treat TestSqlQueries as an actual class, it's more of
# a way to group test functions together. Instead I can
# create a fixture "pg" doing the same thing.
@pytest.fixture
def pg():
    pg = sql_queries.postgresConnection()
    pg.search_term = 'pineapple-shrimp-noodle-bowls'
    pg.fuzzy_search_term = 'chicken'
    pg.random_search_term = r'124 9i2oehf lkajliojk>,/1?/"490_£"'
    pg.phrase_search_term = 'vegan cookies'
    pg.search_column = 'combined_tsv'
    pg.sql_inj1 = "''; SELECT true; --"
    pg.sql_inj2 = "'; SELECT true; --"
    return pg


class TestSqlQueries:

    def test_connect(self, pg):
        assert pg.conn.closed == 0

    def test_fuzzy_search(self, pg):
        # normal querries
        result = pg.fuzzy_search(pg.fuzzy_search_term, N=2)  # substr of "url"
        assert len(result) == 2
        assert pg.fuzzy_search_term in result[0][2]
        assert pg.fuzzy_search_term in result[1][2]
        result = pg.fuzzy_search(pg.random_search_term, N=2)  # not in "url"
        assert len(result) == 2

        # sql injections
        assert len(pg.fuzzy_search(pg.sql_inj1, N=2)) == 2
        assert len(pg.fuzzy_search(pg.sql_inj2, N=2)) == 2
        with pytest.raises(ps.errors.InvalidTextRepresentation):
            pg.fuzzy_search(pg.fuzzy_search_term, N=pg.sql_inj1)

    def test_phrase_search(self, pg):
        # normal querries
        result = pg.phrase_search(pg.search_column, pg.phrase_search_term, N=2)
        assert len(result) == 2

        # sql injections
        assert len(pg.phrase_search(pg.search_column, pg.sql_inj1, N=2)) == 0
        assert len(pg.phrase_search(pg.search_column, pg.sql_inj2, N=2)) == 0
        with pytest.raises(ps.errors.InvalidTextRepresentation):
            pg.phrase_search(pg.search_column, pg.phrase_search_term,
                             N=pg.sql_inj1)

    def test_free_search(self, pg):
        # normal querries
        result = pg.free_search(pg.phrase_search_term, N=2)
        assert len(result) >= 2

        # sql injections
        assert len(pg.free_search(pg.sql_inj1, N=2)) == 2
        assert len(pg.free_search(pg.sql_inj2, N=2)) == 2
        with pytest.raises(ps.errors.InvalidTextRepresentation):
            pg.free_search(pg.phrase_search_term, N=pg.sql_inj1)

    def test_query_content_similarity_ids(self, pg):
        # normal querries
        result = pg.query_content_similarity_ids(pg.search_term)
        assert result[0:10] == (563, 2326, 343, 19957, 927,
                                141, 426, 2034, 13011, 29678)

        # sql injections
        with pytest.raises(IndexError):
            pg.query_content_similarity_ids(pg.sql_inj1)
        with pytest.raises(IndexError):
            pg.query_content_similarity_ids(pg.sql_inj2)

    def test_query_content_similarity(self, pg):
        # normal querries
        result = pg.query_content_similarity(pg.search_term)
        assert result[0:10] == (1.0, 0.452267, 0.43301266, 0.4166667,
                                0.41602513, 0.41247895, 0.41247895,
                                0.4082483, 0.40089187, 0.3952847)

        # sql injections
        with pytest.raises(IndexError):
            pg.query_content_similarity(pg.sql_inj1)
        with pytest.raises(IndexError):
            pg.query_content_similarity(pg.sql_inj2)

    def test_query_similar_recipes(self, pg):
        CS_ids = pg.query_content_similarity_ids(pg.search_term)
        result = pg.query_similar_recipes(CS_ids[0:2])
        assert len(result) == 2

    def test_content_based_search(self, pg):
        result = pg.content_based_search(pg.search_term)
        assert result.iloc[0]['similarity'] == 1.
        assert result.iloc[1]['similarity'] > 0.45

    def test_search_recipes(self, pg):
        # TODO what is being tested here?
        pg.content_based_search(pg.search_term)
        pg.fuzzy_search(pg.fuzzy_search_term)


# eof
```

Making the website more secure

Oct 07 2020

There is certainly still a lot I have to learn about website security and possible different attacks. But the following should make my website quite a bit more secure. Importantly, I believe I can be reasonable certain that visitors data will be handled securely. If anyone disagrees, please let me know!

SQL injection attacks
To prevent SQL injections attacks I used psycopg2's `sql.SQL` module. Specifically, all inputs into a query are parsed by the `.format()` portion of the call with either `sql.Identifier` or `sql.Literal`. See here for more information (last portion of the blog post).

Cross site scripting (XSS)
To prevent XSS (injection of malicious javascript into my website, e.g. via a form) and a host of other security concerns, I opted to use flask_talisman. By setting content security policy (CSP) to disallow inline JS and inline styling (CSS), and removing (almost) all inline JS and styling, XSS should not be possible anymore. The only exception of where I still use inline JS is in results.html, which handles rendering of the rating, sustainability and similarity scales, as well as the dropdown menu for sorting and the Altair figure. To secure it anyway I used a nonce token.

HTTPS (SSL) The website is now also forced to only accept secure connections (no HTTP), which is only enabled in production, because I could not get HTTPS to work on my local machine. I believe this might be an issue with WSL. To be able to test a new update manually without deploying it to production I created a staging environment on Heroku.

Cross site request forgery (CSRF) For this I simply followed the instructions of the official Flask documentation. I called `CSRFProtect()` on the application object and included `{{ form.csrf_token }}` with each Flask form and `<input type="hidden" name="csrf_token" value="{{ csrf_token() }}"/>` with non-flask forms.

That's all for now. Thanks for reading. And feel free to reach out if you are interested in the project, have questions, ideas for improvement or want to collaborate (sbuergers at gmail dot com).