CS5600/6600: F20: Intelligent Systems
Assignment 12
Clustering Texts with K Means and Writing GPS Operators

Vladimir Kulyukin
Department of Computer Science
Utah State University

December 5, 2020

1 Learning Objectives

1. K Means in Textual Information Retrieval
2. Reduction of Search Space with Clustering
3. Means-Ends Analysis
4. General Problem Solver (GPS)
5. Writing GPS Operators

Introduction

This is the last assignment for CS5600/6600: F20. There's no paper analysis –. only two coding problems. The first problem wraps up our exposure to numerical NLP methods from Lectures 20 and 21 and from Problem 2 of Assignment 11. The second problem will give you an opportunity to write a few operators for GPS, a classical AI planner.

Before you work on Problem 1, you may want to review the pdfs for CS5600/6600: F20: Lecture 20 and CS5600/6600: F20: Lecture 21 in Canvas/Announcements and re-read Problem 2 in Assignment 11. I've used the terms and concepts from these documents in this assignment without re-defining or explaining them.

For Problem 2, you may want to review the pdf of CS5600/6600: F20: Lecture 22 in Canvas/Announcements. The materials for Lecture 22 include the article "GPS, A Program that Simulates Human Thought" by A. Newell and P. Simon. This article has influenced multiple generations of AI planning researchers and created an intellectual framework that is still with us today. GPS was a major benchmark in symbolic AI planning. Its knowledge engineering methodology of desigining problem-specific operators and using a general problem solving strategy (e.g., means-ends analysis) to manipulate them is still used in many AI planners.

Problem 1 ($2\frac{1}{2}$ points)

Recall that in Problem 2 of Assignment 11 we built a mini search engine to retrieve posts from 20 USENET newsgroups from `sklearn.datasets`. Specifically, we implemented the function `find_top_n_posts()` that takes a vectorizer object, a user query string, a feat mat computed with the vectorizer, a distance function, and a `top_n` keyword parameter that specifies how many posts the search should return.

In this problem, we'll optimize the engine's retrieval by distributing it over data clusers, each of which contains a subset of the USENET data. We can subsequently distribute each cluster on a different computer (or a cluster of computers) to parallelize cluster search, but we won't dive into it here, because that's distributed computing, not AI.

Let's cluster the USENET data with K Means. In Assignment 11, we defined a vectorizer that uses stemming and word frequencies. In this assignment, we'll use a different vectorizer that uses stemming with tfidf. The source below is in `usenet_kmeans.py`.

```
from sklearn.feature_extraction.text import TfidfVectorizer
import nltk.stem
english_stemmer = nltk.stem.SnowballStemmer('english')
class StemmedTfidfVectorizer(TfidfVectorizer):
    def build_analyzer(self):
        analyzer = super(TfidfVectorizer, self).build_analyzer()
        return lambda doc: (english_stemmer.stem(w) for w in analyzer(doc))
```

The file usenet_kmeans.py contains the two functions below for loading and normalizing the USENET data with the tfidf vectorizer.

```
def load_usenet_data():
    usenet_data = sklearn.datasets.fetch_20newsgroups()
    assert len(usenet_data.target_names) == 20
    print('USENET data loaded...')
    pickle_usenet_data(usenet_data, 'usenet_data.pck')
    return usenet_data

def tfidf_vocab_normalize_usenet_data(usenet_data):
    vectorizer = StemmedTfidfVectorizer(min_df=10,
                                        stop_words='english',
                                        decode_error='ignore')
    feat_mat = vectorizer.fit_transform(usenet_data.data)
    return vectorizer, feat_mat
```

Let's load and normalize the data.

```
>>> from usenet_kmeans import *
>>> usenet_data = load_usenet_data()
>>> usenet_vectorizer, usenet_feat_mat = tfidf_vocab_normalize_usenet_data(usenet_data)
```

We can now create a K Means object and fit it to the data. I've chosen to cluster the data into 50 clusters. It's a completely arbitrary number. You may want to play with different numbers of clusters and investigate their impact on performance and accuracy. The inertia coefficients below are estimates of how far the cluster centroids move from iteration to iteration. When the centroids stop moving, K Means stops. As my output below shows, my K Means object converged after 25 iterations.

```
>>> km = KMeans(n_clusters=50, n_init=1, verbose=1)
>>> km.fit(usenet_feat_mat)
Initialization complete
Iteration  0, inertia 19549.561
Iteration  1, inertia 10582.753
Iteration  2, inertia 10485.661
Iteration  3, inertia 10446.314
Iteration  4, inertia 10426.088
Iteration  5, inertia 10410.384
Iteration  6, inertia 10395.468
Iteration  7, inertia 10384.818
Iteration  8, inertia 10378.678
Iteration  9, inertia 10373.442
Iteration 10, inertia 10366.987
Iteration 11, inertia 10363.506
Iteration 12, inertia 10361.513
Iteration 13, inertia 10359.792
Iteration 14, inertia 10358.958
Iteration 15, inertia 10358.170
Iteration 16, inertia 10357.828
Iteration 17, inertia 10357.626
Iteration 18, inertia 10357.413
Iteration 19, inertia 10357.292
Iteration 20, inertia 10357.223
Iteration 21, inertia 10357.157
Iteration 22, inertia 10357.108
Iteration 23, inertia 10357.068
Iteration 24, inertia 10357.064
Converged at iteration 24: center shift 0.000000e+00 within tolerance 8.320205e-09
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=50, n_init=1, n_jobs=None, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=1)
```

You may want to pickle all these objects for subsequent debugging and experiments if you decide to leave the assignment and come back to it after working on something else. These pickling utils are in `usenet_kmeans.py`.

```
>>> pickle_usenet_data(usenet_data, 'usenet_data.pck')
>>> pickle_usenet_kmeans(km, 'tfidf_kmeans.pck')
>>> pickle_usenet_vectorizer(usenet_vectorizer, 'usenet_tfidf_vectorizer.pck')
>>> pickle_usenet_feat_mat(usenet_feat_mat, 'usenet_tfidf_feat_mat.pck')
```

You may also want to use `fit_and_pickle_tfidf_kmeans(num_clusters)` that does all the loading, vectorizing, clustering, and pickling for you in one batch. You just need to specify the number of clusters.

Given a user query string, we use the same vectorizer to compute its feat mat that we used to normalize the texts of the posts.

```
>>> user_query = 'disk drive problems'
>>> user_query_feat_mat = usenet_vectorizer.transform([user_query]).getrow(0).toarray()
```

Once we have the user query's feat mat, we can use the fitted K Means object to get the top cluster of the user query's feat mat. Basically, K Means returns the number of the cluster (aka the cluster's label) whose centroid is closest to the user query's feat mat. There's nothing that prevents us from returning top n clusters and then searching each of them. Nor is there anything that prevents us from clustering each of the clusters in turn.

```
>>> user_query_top_cluster_label = km.predict(user_query_feat_mat)[0]
>>> print(user_query_top_cluster_label)
15
```

So, cluster 15 happens to be closest to the user query. Let's find the indices of the posts in the USENET data whose feat mats are in the same cluster (i.e., have the same cluster label).

```
>>> posts_in_same_cluster = (km.labels_ == user_query_top_cluster_label).nonzero()[0]
>>> len(posts_in_same_cluster)
197
>>> posts_in_same_cluster[:5]
array([259, 302, 426, 511, 578])
```

Recall that the USENET dataset contains 11,314 USENET posts. By using K Means to cluster the USENET data and then to match the user query against the clusters' centroids instead of the original posts, we managed to reduce our search space to 197 posts. That's a reduction of more than 98%! Of course, the flip side is that we may lose in terms of recall and precision, because we no longer match the user query against *every* text in the dataset. In datasets with millions of texts, however, clustering becomes a necessity.

We need a feat mat metric that computes the distance index between two feat mats. Here's one that computes the euclidean distance.

```
import scipy as sp
def feat_mat_dist(fm1, fm2):
    return sp.linalg.norm(fm1 - fm2)
```

We can use this distance function to compute the distance between the user query's feat mat and the feat mat of post 511 (or any other post with a legal index). If we want to read the actual text of 511, we do `print(usenet_data.data[511])`.

```
>>> feat_mat_dist(user_query_feat_mat, usenet_feat_mat[511])
1.2686133498465961
```

All the tools are now in place for you to implement the function

```
find_top_n_posts_with_kmeans(usenet_vectorizer,
                             usenet_km,
                             user_query,
                             usenet_data,
                             usenet_feat_mat,
                             dist_fun, top_n=5).
```

This function takes a vectorizer object (`usenet_vectorizer`), a fitted K Means object (`usenet_km`), a user query string (`user_query`), usenet data (`usenet_data` is the sparse matrix returned by `load_usenet_data()`), usenet feat mat (`usenet_feat_mat` is the feat mat of `usenet_data`), a distance function `dist_fun` (e.g., `feat_mat_dist`), and the `top_n` parameter that specifies how many top usenet posts to return. The logic of the function consists of 6 steps.

1. Vectorize the user query with the vectorizer to obtain its feat mat;
2. Use the K Means object to predict the top cluster's label of the user query's feat mat;
3. Find the indices of the posts in the same cluster with the user query's feat mat;
4. Go through the indices of the posts in the same cluster and and apply the distance function to the user query's feat mat and the feat mat of the post at a given index in `usenet_feat_mat` and save the 2-tuple with the distance and the post's text in a list of matches;
5. Sort the list of matches by distance from smallest to largest;
6. Return the `top_n` posts.

I wrote the function `test_kmeans()` in `usenet_kmeans.py` for you to test your implementation. This function unpickles the necessary objects, calls `find_top_n_posts_with_kmeans`, and then prints the distance and text of each of the `top_n` posts.

Problem 2 ($2\frac{1}{2}$ points)

In this problem, we'll solve a famous planning problem with the GPS system. Working on this problem will give you some experience of designing problem-specific operators. The zip has two files – `auxfuns.lisp` and `gps.lisp`. The file `auxfuns.lisp` contains some auxiliary functions and the file `gps.lisp` is a Lisp implementation of the GPS planner.

Loading and Running GPS

Change your directory to where you saved `gps.lisp` and `auxfuns.lisp`, fire up your Lisp, and load `gps.lisp` into it.

```
> (load "gps.lisp")
T
```

In the file `gps.lisp`, the variable `*school-ops*` is a list of operators we developed and analyzed in class to solve the son-at-school problem.

```
(defparameter *school-ops*
  (list
    (make-op :action 'drive-son-to-school
         :preconds '(son-at-home car-works)
         :add-list '(son-at-school)
         :del-list '(son-at-home))
    (make-op :action 'shop-installs-battery
         :preconds '(car-needs-battery shop-knows-problem
                     shop-has-money)
         :add-list '(car-works))
    (make-op :action 'tell-shop-problem
         :preconds '(in-communication-with-shop)
         :add-list '(shop-knows-problem))
    (make-op :action 'telephone-shop
         :preconds '(know-phone-number)
         :add-list '(in-communication-with-shop))
    (make-op :action 'look-up-number
         :preconds '(have-phone-book)
         :add-list '(know-phone-number))
    (make-op :action 'give-shop-money
         :preconds '(have-money)
         :add-list '(shop-has-money)
         :del-list '(have-money))))
```

Let's use these operators to solve the son-at-school problem with GPS. We start by defining the initial state of the world for the GPS planner and the goal for the planner to achieve.

```
> (setf world-state-1 '(son-at-home car-needs-battery have-money
                        have-phone-book))
(SON-AT-HOME CAR-NEEDS-BATTERY HAVE-MONEY HAVE-PHONE-BOOK)
> (setf goal-1 '(son-at-school))
(SON-AT-SCHOOL)
```

Now let's apply GPS to this problem by first telling the system to use `*school-ops*` and then applying the system to the initial state of the world and the goal.

```
> (use *school-ops*)
6
> (gps world-state-1 goal-1)
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
 (EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
 (EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
```

The first call `(use *school-ops*)` returns the number of operators in the list of operators the system is asked to use. As you can see from the second call, GPS returns a plan for the agent to follow. The plan consists of looking up the auto shop's number, phoning the shop, telling the shop about the battery problem, giving the shop the money, having the shop install the battery, and driving the son to school. If you need to save the plan, you can do it by saving it in a variable.

```
> (setf son-at-school-plan (gps world-state-1 goal-1))
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
 (EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
 (EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
> son-at-school-plan
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
 (EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
 (EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
```

If you want to trace how GPS solves the problem step by step, you can call the function `trace-gps` and then call the function `gps` on the world's state and the goal to see the details of the means-ends analysis at work. As we discussed in class and as shown below, GPS works by recursively satisfying the preconditions of each operator so that it can be applied to reduce the differences between the current state of the world and the desired state of the world.

```
> (trace-gps)
(:GPS)
> (gps world-state-1 goal-1)
Goal: SON-AT-SCHOOL
Consider: DRIVE-SON-TO-SCHOOL
  Goal: SON-AT-HOME
  Goal: CAR-WORKS
  Consider: SHOP-INSTALLS-BATTERY
    Goal: CAR-NEEDS-BATTERY
    Goal: SHOP-KNOWS-PROBLEM
    Consider: TELL-SHOP-PROBLEM
      Goal: IN-COMMUNICATION-WITH-SHOP
      Consider: TELEPHONE-SHOP
        Goal: KNOW-PHONE-NUMBER
        Consider: LOOK-UP-NUMBER
          Goal: HAVE-PHONE-BOOK
        Action: LOOK-UP-NUMBER
      Action: TELEPHONE-SHOP
    Action: TELL-SHOP-PROBLEM
    Goal: SHOP-HAS-MONEY
    Consider: GIVE-SHOP-MONEY
      Goal: HAVE-MONEY
    Action: GIVE-SHOP-MONEY
  Action: SHOP-INSTALLS-BATTERY
```

```
Action: DRIVE-SON-TO-SCHOOL
((START) (EXECUTE LOOK-UP-NUMBER) (EXECUTE TELEPHONE-SHOP)
 (EXECUTE TELL-SHOP-PROBLEM) (EXECUTE GIVE-SHOP-MONEY)
 (EXECUTE SHOP-INSTALLS-BATTERY) (EXECUTE DRIVE-SON-TO-SCHOOL))
```

If you want to turn the tracer off, do

```
> (untrace-gps)
```

Monkey and Bananas

The monkey and bananas problem is a classic AI planning problem, as classic as Sussman's Anomaly. AI planners are tested on it as they are being developed. Imagine the following situation. A hungry monkey is standing at the doorway to a room. In the middle of the room there is a bunch of bananas suspended from the ceiling by a rope, well out of the monkey's reach. There is a chair near the door, which the monkey can push. The chair is tall enough for the monkey to get the bananas after he climbs on it.

Your job is to develop a set of operators for the monkey to quench his hunger and save your operators in the variable `*banana-ops*` in `gps.lisp`. Below is the output of my solution to the problem. Of course, your solution may be different in that your knowledge representation and your operators may be different. Whatever your final plan is, the goal `(not-hungry)` must be achieved.

```
> (use *banana-ops*)
6
> (setf world-state-2 '(at-door on-floor has-ball hungry chair-at-door))
(AT-DOOR ON-FLOOR HAS-BALL HUNGRY CHAIR-AT-DOOR)
> (setf goal-2 '(not-hungry))
(NOT-HUNGRY)
> (gps world-state-2 goal-2)
Goal: NOT-HUNGRY
Consider: EAT-BANANAS
  Goal: HAS-BANANAS
  Consider: GRASP-BANANAS
    Goal: AT-BANANAS
    Consider: CLIMB-ON-CHAIR
      Goal: CHAIR-AT-MIDDLE-ROOM
      Consider: PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM
        Goal: CHAIR-AT-DOOR
        Goal: AT-DOOR
      Action: PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM
      Goal: AT-MIDDLE-ROOM
      Goal: ON-FLOOR
    Action: CLIMB-ON-CHAIR
    Goal: EMPTY-HANDED
    Consider: DROP-BALL
      Goal: HAS-BALL
    Action: DROP-BALL
  Action: GRASP-BANANAS
Action: EAT-BANANAS
((START) (EXECUTE PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM)
 (EXECUTE CLIMB-ON-CHAIR) (EXECUTE DROP-BALL)
 (EXECUTE GRASP-BANANAS) (EXECUTE EAT-BANANAS))
```

What to Submit?

1. `usenet_kmeans.py` with your implementation of `find_top_n_posts_with_kmeans`;
2. `gps.lisp` with your operators in `*banana-ops*`; save in the comments at the beginning of `gps.lisp` the plan the GPS found with your set of operators.

Happy Hacking!