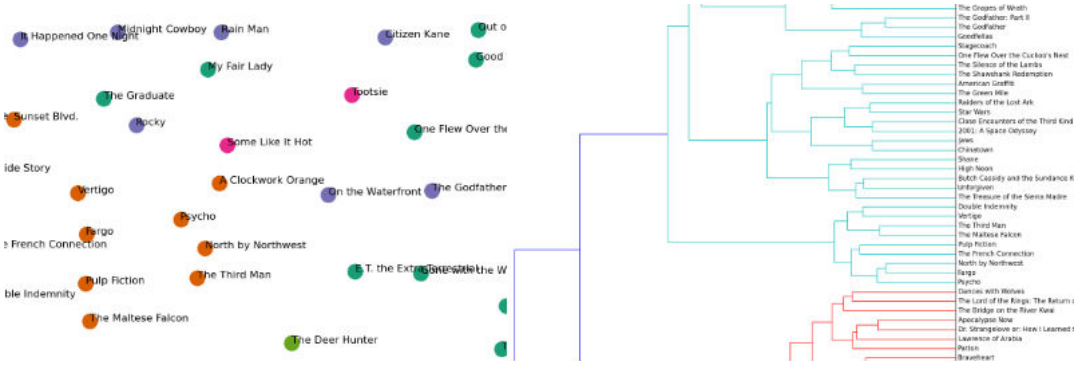
Document Clustering with Python



In this guide, I will explain how to cluster a set of documents using Python. My motivating example is to identify the latent structures within the synopses of the top 100 films of all time (per an IMDB list). See the original post (http://www.brandonrose.org/top100) for a more detailed discussion on the example. This guide covers:

- tokenizing and stemming each synopsis
- transforming the corpus into vector space using tf-idf (http://en.wikipedia.org/wiki/Tf%E2%80%93idf)
- calculating cosine distance between each document as a measure of similarity
- clustering the documents using the k-means algorithm (http://en.wikipedia.org/wiki/K-means_clustering)
- using multidimensional scaling (http://en.wikipedia.org/wiki/Multidimensional_scaling) to reduce dimensionality within the corpus
- plotting the clustering output using matplotlib (http://matplotlib.org/) and mpld3 (http://mpld3.github.io/)
- conducting a hierarchical clustering on the corpus using Ward clustering (http://en.wikipedia.org/wiki/Ward%27s_method)
- plotting a Ward dendrogram
- topic modeling using Latent Dirichlet Allocation (LDA) (http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation)

Note that my github repo (https://github.com/brandomr/document_cluster) for the whole project is available. The 'cluster_analysis' workbook is fully functional; the 'cluster_analysis_web' workbook has been trimmed down for the purpose of creating this walkthrough. Feel free to download the repo and use 'cluster_analysis' to step through the guide yourself.

If you have any questions for me, feel free to reach out on Twitter to @brandonmrose (https://twitter.com/brandonmrose)

Contents

But first, I import everything I am going to need up front

In [4]:
```
import numpy as np
import pandas as pd
import nltk
import re
import os
import codecs
from sklearn import feature_extraction
import mpld3
```

For the purposes of this walkthrough, imagine that I have 2 primary lists:

- *'titles'*: the titles of the films in their rank order
- *'synopses'*: the synopses of the films matched to the 'titles' order

In the full workbook that I posted to github you can walk through the import of these lists, but for brevity just keep in mind that for the rest of this walk-through I will focus on using these two lists. Of primary importance is the ***'synopses'*** list; *'titles'* is mostly used for labeling purposes.

In [17]:
```
print titles[:10] #first 10 titles
```

```
['The Godfather', 'The Shawshank Redemption', "Schindler's List", 'Raging Bull', 'Casablanca', "One Flew Over the Cuckoo's Nest", 'Gone
 with the Wind', 'Citizen Kane', 'The Wizard of Oz', 'Titanic']
```

In [21]:
```
print synopses[0][:200] #first 200 characters in first synopses (for 'The Godfather')
print
print
```

```
Plot [edit] [ [ edit edit ] ]
 On the day of his only daughter's wedding, Vito Corleone hears requests in his role as the Godfather, the Don of a New York crime famil
y. Vito's youngest son, Michael,
```

Stopwords, stemming, and tokenizing

This section is focused on defining some functions to manipulate the synopses. First, I load NLTK's (http://www.nltk.org/) list of English stop words. Stop words (http://en.wikipedia.org/wiki/Stop_words) are words like "a", "the", or "in" which don't convey significant meaning. I'm sure there are much better explanations of this out there.

In [23]:
```
# load nltk's English stopwords as variable called 'stopwords'
stopwords = nltk.corpus.stopwords.words('english')
```

In [51]:
```
print stopwords[:10]
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', 'your']
```

Next I import the Snowball Stemmer (http://snowball.tartarus.org/) which is actually part of NLTK. Stemming (http://en.wikipedia.org/wiki/Stemming) is just the process of breaking a word down into its root.

In [32]:
```
# load nltk's SnowballStemmer as variabled 'stemmer'
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer("english")
```

Below I define two functions:

- *tokenize_and_stem*: tokenizes (splits the synopsis into a list of its respective words (or tokens) and also stems each token
- *tokenize_only*: tokenizes the synopsis only

I use both these functions to create a dictionary which becomes important in case I want to use stems for an algorithm, but later convert stems back to their full words for presentation purposes. Guess what, I do want to do that!

In [33]:
```
# here I define a tokenizer and stemmer which returns the set of stems in the text that it is passed

def tokenize_and_stem(text):
    # first tokenize by sentence, then by word to ensure that punctuation is caught as it's own token
    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)]
    filtered_tokens = []
    # filter out any tokens not containing letters (e.g., numeric tokens, raw punctuation)
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    stems = [stemmer.stem(t) for t in filtered_tokens]
    return stems


def tokenize_only(text):
    # first tokenize by sentence, then by word to ensure that punctuation is caught as it's own token
    tokens = [word.lower() for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent)]
    filtered_tokens = []
    # filter out any tokens not containing letters (e.g., numeric tokens, raw punctuation)
    for token in tokens:
        if re.search('[a-zA-Z]', token):
            filtered_tokens.append(token)
    return filtered_tokens
```

Below I use my stemming/tokenizing and tokenizing functions to iterate over the list of synopses to create two vocabularies: one stemmed and one only tokenized.

In [49]:
```
#not super pythonic, no, not at all.
#use extend so it's a big flat list of vocab
totalvocab_stemmed = []
totalvocab_tokenized = []
for i in synopses:
    allwords_stemmed = tokenize_and_stem(i) #for each item in 'synopses', tokenize/stem
    totalvocab_stemmed.extend(allwords_stemmed) #extend the 'totalvocab_stemmed' list

    allwords_tokenized = tokenize_only(i)
    totalvocab_tokenized.extend(allwords_tokenized)
```

Using these two lists, I create a pandas DataFrame with the stemmed vocabulary as the index and the tokenized words as the column. The benefit of this is it provides an efficient way to look up a stem and return a full token. The downside here is that stems to tokens are one to many: the stem 'run' could be associated with 'ran', 'runs', 'running', etc. For my purposes this is fine--I'm perfectly happy returning the first token associated with the stem I need to look up.

In [48]:
```
vocab_frame = pd.DataFrame({'words': totalvocab_tokenized}, index = totalvocab_stemmed)
print 'there are ' + str(vocab_frame.shape[0]) + ' items in vocab_frame'
```

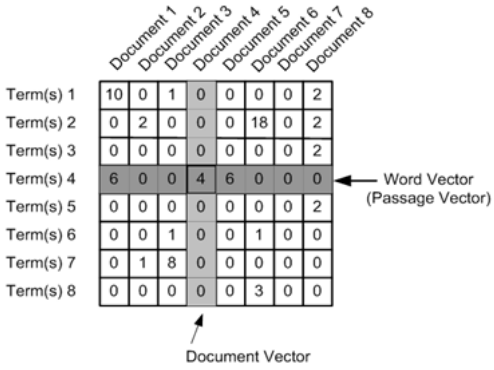
```
there are 312209 items in vocab_frame
```

You'll notice there is clearly some repetition here. I could clean it up, but there are only 312209 items in the DataFrame which isn't huge overhead in looking up a stemmed word based on the stem-index.

In [50]:
```
print vocab_frame.head()
print
print
print
print
```

```
     words
plot  plot
edit  edit
edit  edit
edit  edit
on      on
```

Tf-idf and document similarity



Here, I define term frequency-inverse document frequency (tf-idf) vectorizer parameters and then convert the *synopses* list into a tf-idf matrix.

To get a Tf-idf matrix, first count word occurrences by document. This is transformed into a document-term matrix (dtm). This is also just called a term frequency matrix. An example of a dtm is here at right.

Then apply the term frequency-inverse document frequency weighting: words that occur frequently within a document but not frequently within the corpus receive a higher weighting as these words are assumed to contain more meaning in relation to the document.

A couple things to note about the parameters I define below:

- max_df: this is the maximum frequency within the documents a given feature can have to be used in the tfi-idf matrix. If the term is in greater than 80% of the documents it probably cares little meanining (in the context of film synopses)
- min_idf: this could be an integer (e.g. 5) and the term would have to be in at least 5 of the documents to be considered. Here I pass 0.2; the term must be in at least 20% of the document. I found that if I allowed a lower min_df I ended up basing clustering on names--for example "Michael" or "Tom" are names found in several of the movies and the synopses use these names frequently, but the names carry no real meaning.
- ngram_range: this just means I'll look at unigrams, bigrams and trigrams. See n-grams (http://en.wikipedia.org/wiki/N-gram)

In [54]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

#define vectorizer parameters
tfidf_vectorizer = TfidfVectorizer(max_df=0.8, max_features=200000,
                                   min_df=0.2, stop_words='english',
                                   use_idf=True, tokenizer=tokenize_and_stem, ngram_range=(1,3))

%time tfidf_matrix = tfidf_vectorizer.fit_transform(synopses) #fit the vectorizer to synopses

print(tfidf_matrix.shape)
```

```
CPU times: user 29.1 s, sys: 468 ms, total: 29.6 s
Wall time: 37.8 s
(100, 563)
```

terms is just a list of the features used in the tf-idf matrix. This is a vocabulary

In [55]:
```
terms = tfidf_vectorizer.get_feature_names()
```

dist is defined as 1 - the cosine similarity of each document. Cosine similarity is measured against the tf-idf matrix and can be used to generate a measure of similarity between each document and the other documents in the corpus (each synopsis among the synopses). Subtracting it from 1 provides cosine distance which I will use for plotting on a euclidean (2-dimensional) plane.

Note that with *dist* it is possible to evaluate the similarity of any two or more synopses.

In [57]:
```
from sklearn.metrics.pairwise import cosine_similarity
dist = 1 - cosine_similarity(tfidf_matrix)
print
print
```

K-means clustering

Now onto the fun part. Using the tf-idf matrix, you can run a slew of clustering algorithms to better understand the hidden structure within the synopses. I first chose k-means (http://en.wikipedia.org/wiki/K-means_clustering). K-means initializes with a pre-determined number of clusters (I chose 5). Each observation is assigned to a cluster (cluster assignment) so as to minimize the within cluster sum of squares. Next, the mean of the clustered observations is calculated and used as the new cluster centroid. Then, observations are reassigned to clusters and centroids recalculated in an iterative process until the algorithm reaches convergence.

I found it took several runs for the algorithm to converge a global optimum as k-means is susceptible to reaching local optima.

In [66]:
```
from sklearn.cluster import KMeans

num_clusters = 5

km = KMeans(n_clusters=num_clusters)

%time km.fit(tfidf_matrix)

clusters = km.labels_.tolist()
```

```
CPU times: user 232 ms, sys: 6.64 ms, total: 239 ms
Wall time: 305 ms
```

I use joblib.dump to pickle the model, once it has converged and to reload the model/reassign the labels as the clusters.

In [67]:
```
from sklearn.externals import joblib

#uncomment the below to save your model
#since I've already run my model I am loading from the pickle

#joblib.dump(km,  'doc_cluster.pkl')

km = joblib.load('doc_cluster.pkl')
clusters = km.labels_.tolist()
```

Here, I create a dictionary of titles, ranks, the synopsis, the cluster assignment, and the genre [rank and genre were scraped from IMDB].

I convert this dictionary to a Pandas DataFrame for easy access. I'm a huge fan of Pandas (http://pandas.pydata.org/) and recommend taking a look at some of its awesome functionality which I'll use below, but not describe in a ton of detail.

In [62]:
```
films = { 'title': titles, 'rank': ranks, 'synopsis': synopses, 'cluster': clusters, 'genre': genres }

frame = pd.DataFrame(films, index = [clusters] , columns = ['rank', 'title', 'cluster', 'genre'])
```

In [63]:
```
frame['cluster'].value_counts() #number of films per cluster (clusters from 0 to 4)
```

Out[63]:
```
4    26
0    25
2    21
1    16
3    12
dtype: int64
```

In [64]:
```
grouped = frame['rank'].groupby(frame['cluster']) #groupby cluster for aggregation purposes

grouped.mean() #average rank (1 to 100) per cluster
```

Out[64]:
```
cluster
0          47.200000
1          58.875000
2          49.380952
3          54.500000
4          43.730769
dtype: float64
```

Note that **clusters 4 and 0** have the lowest rank, which indicates that they, on average, contain films that were ranked as "better" on the top 100 list.

Here is some fancy indexing and sorting on each cluster to identify which are the top *n* (I chose *n*=6) words that are nearest to the cluster centroid. This gives a good sense of the main topic of the cluster.

In [69]:
```
from __future__ import print_function

print("Top terms per cluster:")
print()
#sort cluster centers by proximity to centroid
order_centroids = km.cluster_centers_.argsort()[:, ::-1]

for i in range(num_clusters):
    print("Cluster %d words:" % i, end='')

    for ind in order_centroids[i, :6]: #replace 6 with n words per cluster
        print(' %s' % vocab_frame.ix[terms[ind].split(' ')].values.tolist()[0][0].encode('utf-8', 'ignore'), end=',')
    print() #add whitespace
    print() #add whitespace

    print("Cluster %d titles:" % i, end='')
    for title in frame.ix[i]['title'].values.tolist():
        print(' %s,' % title, end='')
    print() #add whitespace
    print() #add whitespace

print()
print()
```

```
Top terms per cluster:

Cluster 0 words: family, home, mother, war, house, dies,

Cluster 0 titles: Schindler's List, One Flew Over the Cuckoo's Nest, Gone with the Wind, The Wizard of Oz, Titanic, Forrest Gump, E.T. t
he Extra-Terrestrial, The Silence of the Lambs, Gandhi, A Streetcar Named Desire, The Best Years of Our Lives, My Fair Lady, Ben-Hur, Do
ctor Zhivago, The Pianist, The Exorcist, Out of Africa, Good Will Hunting, Terms of Endearment, Giant, The Grapes of Wrath, Close Encoun
ters of the Third Kind, The Graduate, Stagecoach, Wuthering Heights,

Cluster 1 words: police, car, killed, murders, driving, house,

Cluster 1 titles: Casablanca, Psycho, Sunset Blvd., Vertigo, Chinatown, Amadeus, High Noon, The French Connection, Fargo, Pulp Fiction,
 The Maltese Falcon, A Clockwork Orange, Double Indemnity, Rebel Without a Cause, The Third Man, North by Northwest,

Cluster 2 words: father, new, york, new, brothers, apartments,

Cluster 2 titles: The Godfather, Raging Bull, Citizen Kane, The Godfather: Part II, On the Waterfront, 12 Angry Men, Rocky, To Kill a Mo
ckingbird, Braveheart, The Good, the Bad and the Ugly, The Apartment, Goodfellas, City Lights, It Happened One Night, Midnight Cowboy, M
r. Smith Goes to Washington, Rain Man, Annie Hall, Network, Taxi Driver, Rear Window,

Cluster 3 words: george, dance, singing, john, love, perform,

Cluster 3 titles: West Side Story, Singin' in the Rain, It's a Wonderful Life, Some Like It Hot, The Philadelphia Story, An American in
 Paris, The King's Speech, A Place in the Sun, Tootsie, Nashville, American Graffiti, Yankee Doodle Dandy,

Cluster 4 words: killed, soldiers, captain, men, army, command,

Cluster 4 titles: The Shawshank Redemption, Lawrence of Arabia, The Sound of Music, Star Wars, 2001: A Space Odyssey, The Bridge on the
 River Kwai, Dr. Strangelove or: How I Learned to Stop Worrying and Love the Bomb, Apocalypse Now, The Lord of the Rings: The Return of
 the King, Gladiator, From Here to Eternity, Saving Private Ryan, Unforgiven, Raiders of the Lost Ark, Patton, Jaws, Butch Cassidy and t
he Sundance Kid, The Treasure of the Sierra Madre, Platoon, Dances with Wolves, The Deer Hunter, All Quiet on the Western Front, Shane,
 The Green Mile, The African Queen, Mutiny on the Bounty,
```

Multidimensional scaling

Here is some code to convert the dist matrix into a 2-dimensional array using multidimensional scaling (http://en.wikipedia.org/wiki/Multidimensional_scaling). I won't pretend I know a ton about MDS, but it was useful for this purpose. Another option would be to use principal component analysis (http://en.wikipedia.org/wiki/Principal_component_analysis).

In [74]:
```
import os  # for os.path.basename

import matplotlib.pyplot as plt
import matplotlib as mpl

from sklearn.manifold import MDS

MDS()

# convert two components as we're plotting points in a two-dimensional plane
# "precomputed" because we provide a distance matrix
# we will also specify `random_state` so the plot is reproducible.
mds = MDS(n_components=2, dissimilarity="precomputed", random_state=1)

pos = mds.fit_transform(dist)  # shape (n_components, n_samples)

xs, ys = pos[:, 0], pos[:, 1]
print()
print()
```

Visualizing document clusters

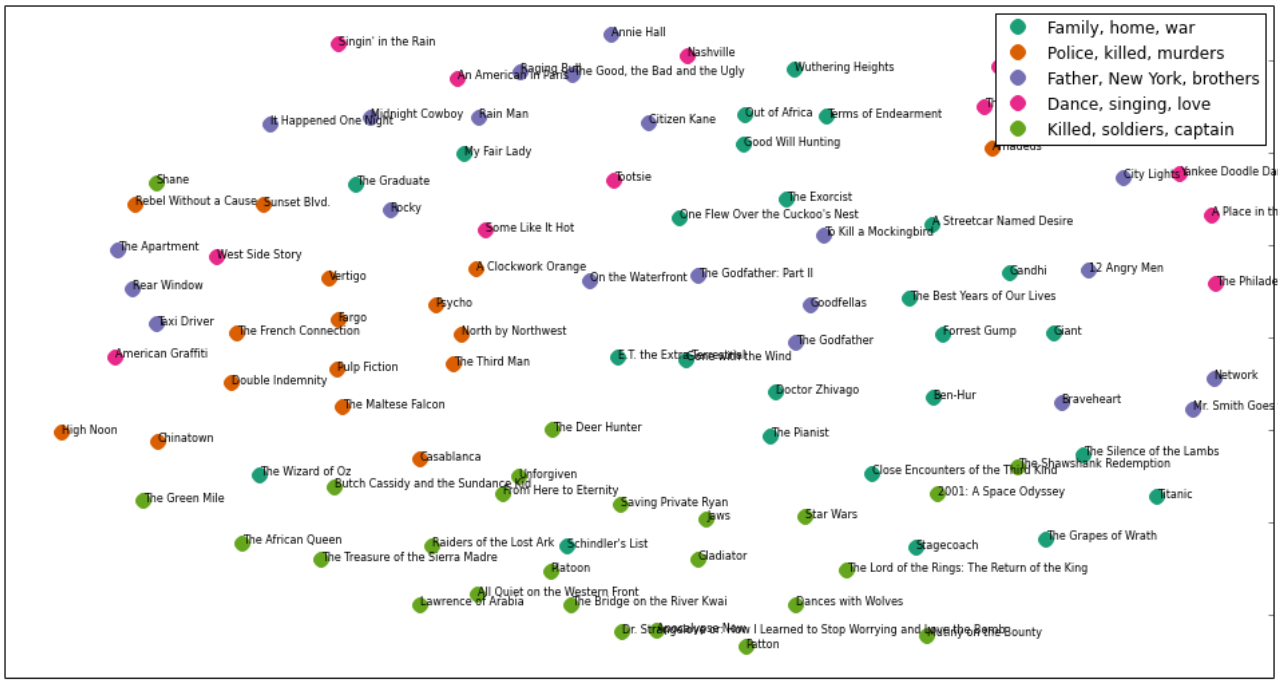
In this section, I demonstrate how you can visualize the document clustering output using matplotlib and mpld3 (a matplotlib wrapper for D3.js).

First I define some dictionaries for going from cluster number to color and to cluster name. I based the cluster names off the words that were closest to each cluster centroid.

In [75]:
```
#set up colors per clusters using a dict
cluster_colors = {0: '#1b9e77', 1: '#d95f02', 2: '#7570b3', 3: '#e7298a', 4: '#66a61e'}

#set up cluster names using a dict
cluster_names = {0: 'Family, home, war',
                 1: 'Police, killed, murders',
                 2: 'Father, New York, brothers',
                 3: 'Dance, singing, love',
                 4: 'Killed, soldiers, captain'}
```

Next, I plot the labeled observations (films, film titles) colored by cluster using matplotlib. I won't get into too much detail about the matplotlib plot, but I tried to provide some helpful commenting.

In [51]:
```
#some ipython magic to show the matplotlib plots inline
%matplotlib inline

#create data frame that has the result of the MDS plus the cluster numbers and titles
df = pd.DataFrame(dict(x=xs, y=ys, label=clusters, title=titles))

#group by cluster
groups = df.groupby('label')


# set up plot
fig, ax = plt.subplots(figsize=(17, 9)) # set size
ax.margins(0.05) # Optional, just adds 5% padding to the autoscaling

#iterate through groups to layer the plot
#note that I use the cluster_name and cluster_color dicts with the 'name' lookup to return the appropriate color/label
for name, group in groups:
    ax.plot(group.x, group.y, marker='o', linestyle='', ms=12,
            label=cluster_names[name], color=cluster_colors[name],
            mec='none')
    ax.set_aspect('auto')
    ax.tick_params(\
        axis= 'x',          # changes apply to the x-axis
        which='both',      # both major and minor ticks are affected
        bottom='off',      # ticks along the bottom edge are off
        top='off',         # ticks along the top edge are off
        labelbottom='off')
    ax.tick_params(\
        axis= 'y',         # changes apply to the y-axis
        which='both',      # both major and minor ticks are affected
        left='off',      # ticks along the bottom edge are off
        top='off',         # ticks along the top edge are off
        labelleft='off')

ax.legend(numpoints=1)  #show legend with only 1 point

#add label in x,y position with the label as the film title
for i in range(len(df)):
    ax.text(df.ix[i]['x'], df.ix[i]['y'], df.ix[i]['title'], size=8)



plt.show() #show the plot

#uncomment the below to save the plot if need be
#plt.savefig('clusters_small_noaxes.png', dpi=200)
```



In [52]:
```
plt.close()
```

The clustering plot looks great, but it pains my eyes to see overlapping labels. Having some experience with D3.js (http://d3js.org/) I knew one solution would be to use a browser based/javascript interactive. Fortunately, I recently stumbled upon mpld3 (https://mpld3.github.io/) a matplotlib wrapper for D3. Mpld3 basically let's you use matplotlib syntax to create web interactives. It has a really easy, high-level API for adding tooltips on mouse hover, which is what I am interested in.

It also has some nice functionality for zooming and panning. The below javascript snippet basicaly defines a custom location for where the zoom/pan toggle resides. Don't worry about it too much and you actually don't need to use it, but it helped for formatting purposes when exporting to the web later. The only thing you might want to change is the x and y attr for the position of the toolbar.

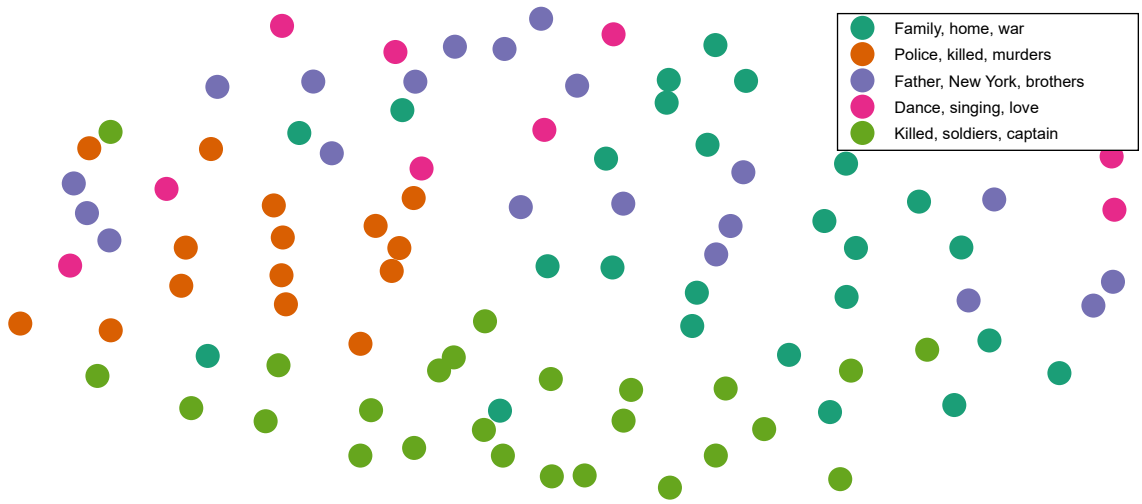
In [77]:
```
#define custom toolbar location
class TopToolbar(mpld3.plugins.PluginBase):
    """Plugin for moving toolbar to top of figure"""

    JAVASCRIPT = """
    mpld3.register_plugin("toptoolbar", TopToolbar);
    TopToolbar.prototype = Object.create(mpld3.Plugin.prototype);
    TopToolbar.prototype.constructor = TopToolbar;
    function TopToolbar(fig, props){
        mpld3.Plugin.call(this, fig, props);
    };

    TopToolbar.prototype.draw = function(){
      // the toolbar svg doesn't exist
      // yet, so first draw it
      this.fig.toolbar.draw();

      // then change the y position to be
      // at the top of the figure
      this.fig.toolbar.toolbar.attr("x", 150);
      this.fig.toolbar.toolbar.attr("y", 400);

      // then remove the draw function,
      // so that it is not called again
      this.fig.toolbar.draw = function() {}
    }
    """
    def __init__(self):
        self.dict_ = {"type": "toptoolbar"}
```

Here is the actual creation of the interactive scatterplot. I won't go into much more detail about it since it's pretty much a straightforward copy of one of the mpld3 examples, though I use a pandas groupby to group by cluster, then iterate through the groups as I layer the scatterplot. Note that relative to doing this with raw D3, mpld3 is much simpler to integrate into your Python workflow. If you click around the rest of my website you'll see that I do love D3, but for basic interactives I will probably use mpld3 a lot going forward.

Note that mpld3 lets you define some custom CSS, which I use to style the font, the axes, and the left margin on the figure.

In [78]:
```
#create data frame that has the result of the MDS plus the cluster numbers and titles
df = pd.DataFrame(dict(x=xs, y=ys, label=clusters, title=titles))

#group by cluster
groups = df.groupby('label')

#define custom css to format the font and to remove the axis labeling
css = """
text.mpld3-text, div.mpld3-tooltip {
  font-family:Arial, Helvetica, sans-serif;
}

g.mpld3-xaxis, g.mpld3-yaxis {
display: none; }

svg.mpld3-figure {
margin-left: -200px;}
"""

# Plot
fig, ax = plt.subplots(figsize=(14,6)) #set plot size
ax.margins(0.03) # Optional, just adds 5% padding to the autoscaling

#iterate through groups to layer the plot
#note that I use the cluster_name and cluster_color dicts with the 'name' lookup to return the appropriate color/label
for name, group in groups:
    points = ax.plot(group.x, group.y, marker='o', linestyle='', ms=18,
                     label=cluster_names[name], mec='none',
                     color=cluster_colors[name])
    ax.set_aspect('auto')
    labels = [i for i in group.title]

    #set tooltip using points, labels and the already defined 'css'
    tooltip = mpld3.plugins.PointHTMLTooltip(points[0], labels,
                                       voffset=10, hoffset=10, css=css)
    #connect tooltip to fig
    mpld3.plugins.connect(fig, tooltip, TopToolbar())

    #set tick marks as blank
    ax.axes.get_xaxis().set_ticks([])
    ax.axes.get_yaxis().set_ticks([])

    #set axis as blank
    ax.axes.get_xaxis().set_visible(False)
    ax.axes.get_yaxis().set_visible(False)


ax.legend(numpoints=1) #show legend with only one dot

mpld3.display() #show the plot

#uncomment the below to export to html
#html = mpld3.fig_to_html(fig)
#print(html)
```

Out[78]:

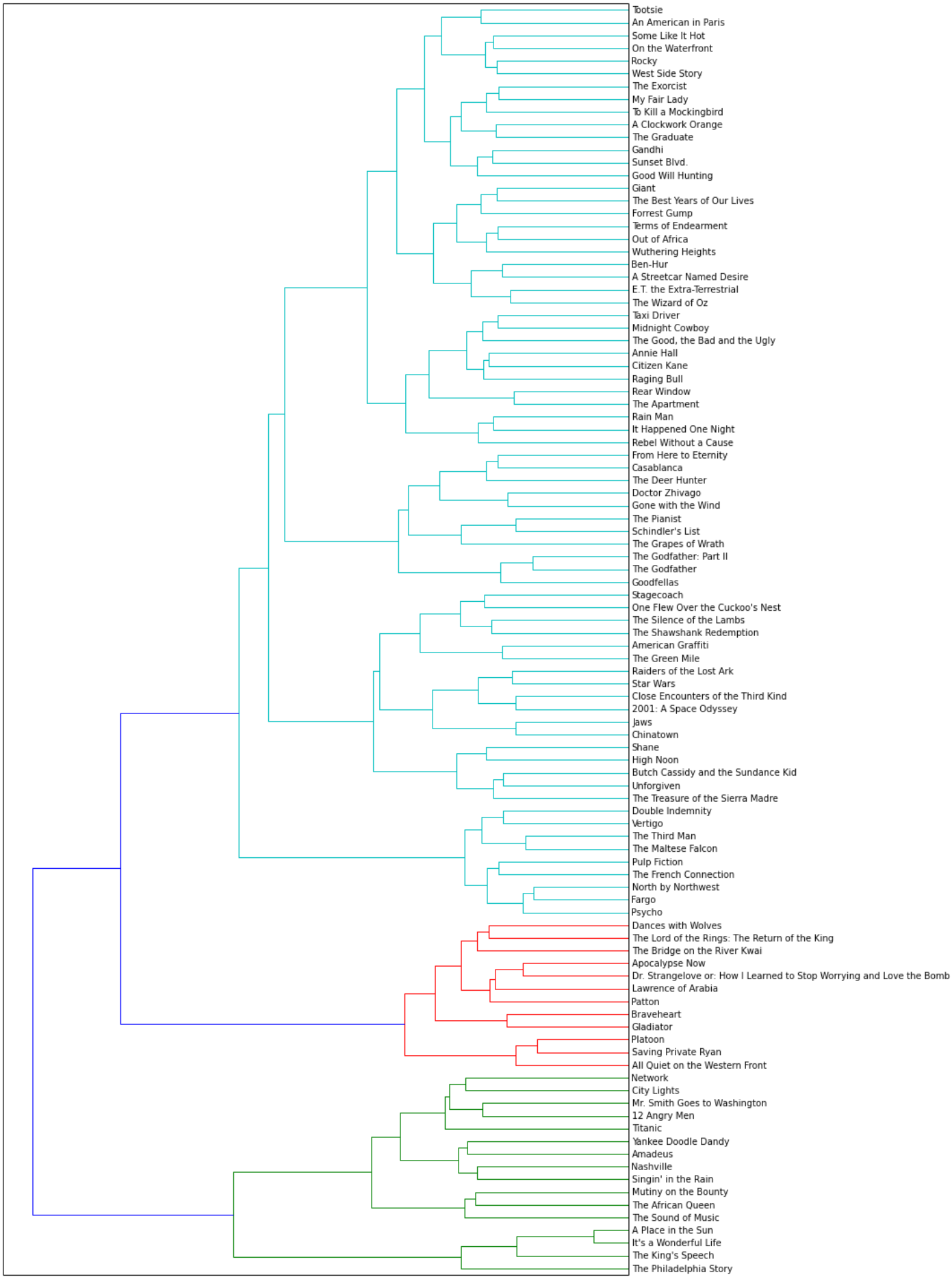


Hierarchical document clustering

Now that I was successfuly able to cluster and plot the documents using k-means, I wanted to try another clustering algorithm. I chose the Ward clustering algorithm (http://en.wikipedia.org/wiki/Ward%27s_method) because it offers hierarchical clustering. Ward clustering is an agglomerative clustering method, meaning that at each stage, the pair of clusters with minimum between-cluster distance are merged. I used the precomputed cosine distance matrix (*dist*) to calclate a linkage_matrix, which I then plot as a dendrogram.

Note that this method returned 3 primary clusters, with the largest cluster being split into about 4 major subclusters. Note that the cluster in red contains many of the "Killed, soldiers, captain" films. *Braveheart* and *Gladiator* are within the same low-level cluster which is interesting as these are probably my two favorite movies.

In [83]:
```
from scipy.cluster.hierarchy import ward, dendrogram

linkage_matrix = ward(dist) #define the linkage_matrix using ward clustering pre-computed distances

fig, ax = plt.subplots(figsize=(15, 20)) # set size
ax = dendrogram(linkage_matrix, orientation="right", labels=titles);

plt.tick_params(\
    axis= 'x',          # changes apply to the x-axis
    which='both',      # both major and minor ticks are affected
    bottom='off',      # ticks along the bottom edge are off
    top='off',         # ticks along the top edge are off
    labelbottom='off')

plt.tight_layout() #show plot with tight layout

#uncomment below to save figure
plt.savefig('ward_clusters.png', dpi=200) #save figure as ward_clusters
```



In [60]:
```
plt.close()
```

Latent Dirichlet Allocation

This section focuses on using Latent Dirichlet Allocation (LDA) (http://en.wikipedia.org/wiki/Latent_Dirichlet_allocation) to learn yet more about the hidden structure within the top 100 film synopses. LDA is a probabilistic topic model that assumes documents are a mixture of topics and that each word in the document is attributable to the document's topics. There is quite a good high-level overview of probabilistic topic models by one of the big names in the field, David Blei, available in the Communications of the ACM here (http://delivery.acm.org/10.1145/2140000/2133826/p77-blei.pdf?ip=68.48.185.120&id=2133826&acc=OPEN&key=4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E6D218144511F3437&CFID=612398453&CFTOKEN=487 Incidentally, Blei was one of the authors of the seminal paper on LDA.

For my implementaiton of LDA, I use the Gensim pacakage (https://radimrehurek.com/gensim/). I'm going to preprocess the synopses a bit differently here, and first I define a function to remove any proper noun.

In [82]:
```
#strip any proper names from a text...unfortunately right now this is yanking the first word from a sentence too.
import string
def strip_proppers(text):
    # first tokenize by sentence, then by word to ensure that punctuation is caught as it's own token
    tokens = [word for sent in nltk.sent_tokenize(text) for word in nltk.word_tokenize(sent) if word.islower()]
    return "".join([" "+i if not i.startswith("'") and i not in string.punctuation else i for i in tokens]).strip()
```

Since the above function is just based on capitalization, it is prone to remove words at the beginning of sentences. So, I wrote the below function using NLTK's part of speech tagger. However, it took way too long to run across all synopses, so I stuck with the above.

In [80]:
```
#strip any proper nouns (NNP) or plural proper nouns (NNPS) from a text
from nltk.tag import pos_tag

def strip_proppers_POS(text):
    tagged = pos_tag(text.split()) #use NLTK's part of speech tagger
    non_propernouns = [word for word,pos in tagged if pos != 'NNP' and pos != 'NNPS']
    return non_propernouns
```

Here I run the actual text processing (removing of proper nouns, tokenization, removal of stop words)

In [83]:
```
from gensim import corpora, models, similarities

#remove proper names
%time preprocess = [strip_proppers(doc) for doc in synopses]

#tokenize
%time tokenized_text = [tokenize_and_stem(text) for text in preprocess]

#remove stop words
%time texts = [[word for word in text if word not in stopwords] for text in tokenized_text]
```

```
CPU times: user 12.9 s, sys: 148 ms, total: 13 s
Wall time: 15.9 s
CPU times: user 15.1 s, sys: 172 ms, total: 15.3 s
Wall time: 19.3 s
CPU times: user 4.56 s, sys: 39.2 ms, total: 4.6 s
Wall time: 5.95 s
```

Below are some Gensim specific conversions; I also filter out extreme words (see inline comment)

In [84]:
```
#create a Gensim dictionary from the texts
dictionary = corpora.Dictionary(texts)

#remove extremes (similar to the min/max df step used when creating the tf-idf matrix)
dictionary.filter_extremes(no_below=1, no_above=0.8)

#convert the dictionary to a bag of words corpus for reference
corpus = [dictionary.doc2bow(text) for text in texts]
```

The actual model runs below. I took 100 passes to ensure convergence, but you can see that it took my machine 13 minutes to run. My chunksize is larger than the corpus so basically all synopses are used per pass. I should optimize this, and Gensim has the capacity to run in parallel. I'll likely explore this further as I use the implementation on larger corpora.

In [85]:
```
%time lda = models.LdaModel(corpus, num_topics=5,
                            id2word=dictionary,
                            update_every=5,
                            chunksize=10000,
                            passes=100)
```

```
CPU times: user 9min 53s, sys: 5.87 s, total: 9min 59s
Wall time: 13min 1s
```

Each topic has a set of words that defines it, along with a certain probability.

In [87]:
```
lda.show_topics()
```

Out[87]:
```
[u'0.006*men + 0.005*kill + 0.004*soldier + 0.004*order + 0.004*patient + 0.004*night + 0.003*priest + 0.003*becom + 0.003*new + 0.003*speech',
 u"0.006*n't + 0.005*go + 0.005*fight + 0.004*doe + 0.004*home + 0.004*famili + 0.004*car + 0.004*night + 0.004*say + 0.004*next",
 u"0.005*ask + 0.005*meet + 0.005*kill + 0.004*say + 0.004*friend + 0.004*car + 0.004*love + 0.004*famili + 0.004*arriv + 0.004*n't",
 u'0.009*kill + 0.006*soldier + 0.005*order + 0.005*men + 0.005*shark + 0.004*attempt + 0.004*offic + 0.004*son + 0.004*command + 0.004*attack',
 u'0.004*kill + 0.004*water + 0.004*two + 0.003*plan + 0.003*away + 0.003*set + 0.003*boat + 0.003*vote + 0.003*way + 0.003*home']
```

Here, I convert the topics into just a list of the top 20 words in each topic. You can see a similar breakdown of topics as I identified using k-means including a war/family topic and a more clearly war/epic topic.

In [86]:
```
topics_matrix = lda.show_topics(formatted=False, num_words=20)
topics_matrix = np.array(topics_matrix)

topic_words = topics_matrix[:,:,1]
for i in topic_words:
    print([str(word) for word in i])
    print()
```

```
['men', 'kill', 'soldier', 'order', 'patient', 'night', 'priest', 'becom', 'new', 'speech', 'friend', 'decid', 'young', 'ward', 'state',
 'front', 'would', 'home', 'two', 'father']

["n't", 'go', 'fight', 'doe', 'home', 'famili', 'car', 'night', 'say', 'next', 'ask', 'day', 'want', 'show', 'goe', 'friend', 'two', 'polic', 'name', 'meet']

['ask', 'meet', 'kill', 'say', 'friend', 'car', 'love', 'famili', 'arriv', "n't", 'home', 'two', 'go', 'father', 'money', 'call', 'polic', 'apart', 'night', 'hous']

['kill', 'soldier', 'order', 'men', 'shark', 'attempt', 'offic', 'son', 'command', 'attack', 'water', 'friend', 'ask', 'fire', 'arriv',
 'wound', 'die', 'battl', 'death', 'fight']

['kill', 'water', 'two', 'plan', 'away', 'set', 'boat', 'vote', 'way', 'home', 'run', 'ship', 'would', 'destroy', 'guilti', 'first', 'attack', 'go', 'use', 'forc']
```