CS109b - Project - Team 14
Milestone 3: Traditional Statistical and Machine Learning Methods
April 18, 2017¶

Introduction

After the exploration of Milestone 1 and the data preparation of Milestone 2, our team implemented two classic machine learning models for Milestone 3. Our data set consisted of approximately 40,000 movies identified in a research set published by MovieLens, which we used to acquire data from IMDb and TMDb. Various team members explored models with elements of this data: movie people, movie overviews, structured metadata, and movie posters. Our choices for predictors and algorithms are described in detail below. On the basis of these choices, model performance was quite satisfactory. The rest of this report will present our models, evaluation strategy and results, and additional thoughts.

Description of Two Models

Our team considered a range of possible models in order to classify movies by genre. Possibilities included the most popular classic machine-learning methods, as well as approaches augmented by natural language processing. We selected two complementary approaches. The first was text mining of the Overview field in the TMDb data. The second was traditional methods applied to people and movie roles. In this way, we explored possibilities with both major data sets that feature in this project. In addition, the two approaches focus on different aspects of the movie industry - human participants vs. descriptive metadata about the products. Ironically, it turned out that the modeling the two domains could be unified to a surprising degree, as narrated below.

1. **Text Mining - Movie Overviews**

   The Overview field in the TMDb data is a rich source of information, with varied content that may include plot description, people roles, industry context, and even genre tags (in a minority of cases). Approaching this data required proceeding in three steps:

   1. Vectorizing each overview on a count basis, thereby creating a document-term matrix. In such a matrix, each row represents an overview and each column represents a unique term in the ‘corpus’ of all overviews. To filter out noise in the overviews, words were stemmed and stopwords were removed. Both stemming and stopword removal was performed in English.

      Vectorization by term-frequency/inverse-document-frequency (TF-IDF) was explored as an alternative, but its performance was similar to that with count vectorization. So the simple count method was preferred.

2. The vocabulary of the collected overviews for our data set included approximately 42,000 terms. This volume is too large for many traditional machine learning techniques, in terms of both computational resources (on personal computers) and the challenges of high-dimensional settings. For this reason, our team chose to reduce dimensionality of the matrix using principal components analysis (PCA). 1. Through some experimentation, we determined that 100 components delivered satisfactory accuracy in prediction, with good computational efficiency.

3. Having a clean, reduced data set in hand, we proceeded with traditional machine learning. We tried a range of techniques that offered reasonable computational efficiency on personal computers, as well as the potential for good prediction accuracy. These techniques included logistic regression, linear discriminant analysis (LDA), quadratic discriminant analysis (QDA), and random forests. Preliminary results showed that both random forests and QDA offered good prediction accuracy similar levels. So the remainder of this report will focus on these two techniques. Details of the implementation can be found in the attached Jupyter notebook.

   It’s worth noting that our modeling used a multi-label approach, rather than a multi-class approach. In preliminary modeling, we found that a multi-class approach with approximately 120 compound classes achieved disappointing accuracy in the range of 2% on the test set. So we switched to a multi-label approach, which achieved satisfactory prediction accuracy. With this approach, we created prediction models for each of the 18 unique genre tags that were shared between IMDb and TMDb. (We plan to carry this multi-label approach forward to the deep learning stage of the project as well.)

2. People Roles in Movies

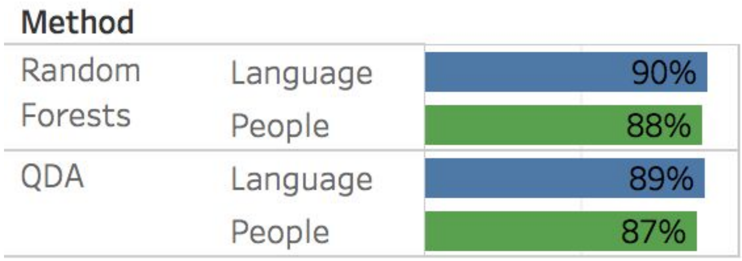
The IMDB dataset contains comprehensive data for the people involved in producing each movie. Our initial hypothesis was that actors, writers , directors, and musicians often specialize in a specific genre. We analysed the people data as “bag of words” in much the same way text data is mined. The approach was as follows:

1. The genres were vectorized into the 19 genres for each movie. These formed our response variables
2. Over two million people were involved in the production of our 36,000 movie dataset. We sought to reduce the problem of high dimensionality. Observing the the majority of people were involved in just a small number of movies. We reasoned that people active in more movies would be the best predictors. Removing people who had been in fewer than ten movies each reduced the number of people down to 419,000.
3. The reduced people dataset was reorganized from a stacked relational database structure, to a cross tabulation with one row per movie with people represented as 419,000 columns with 0 or 1 (1 = in the movie, 0 = not in the movie).
4. Using Principle Component Analysis the predictors were further reduced to a more manageable 100 components.
5. At this point, the data was in a format consistent with the text mining model. We were able to apply the same code base to explore models with QDA, Random Forests, LDA and Logistic regression. Random forests achieved an overall test accuracy of 89%, and QDA performed well with 83%.

During implementation, our insight was that the two modeling domains (people and overviews) were isomorphic, having a similar structure. Just as the overviews could be analyzed into a matrix of documents and terms, the movie people could be analyzed into a matrix of people and movies that they participated in. Once this insight was available, our team shared essentially the same code to pre-process and load the target classes, as well as code to run and score both random-forest and QDA models on the different X data. (Because of the consistency in code and metrics between the models, we reported on two data domains with two model types each, in order to enrich the report.)

Performance of Two Models

Overall accuracy was comparable between the four models we explored. Averaged across all genres, the language model showed a test accuracy of .90 with random forests and .89 with QDA. Averaged across all genres, the people model showed a test accuracy of .89 with random forests and .87 with QDA. So performance with the language model was slightly better. Detailed of the per-genre test accuracy are shown below.



	Random Forests		QDA	
Genre	**Language**	**People**	**Language**	**People**
Action	0.86	0.86	0.85	0.85
Adventure	0.93	0.90	0.93	0.89
Animation	0.97	0.97	0.97	0.96
Comedy	0.70	0.70	0.70	0.66
Crime	0.91	0.85	0.91	0.85
Documentary	0.93	0.94	0.91	0.94
Drama	0.62	0.62	0.60	0.49
Family	0.95	0.94	0.94	0.93
Fantasy	0.96	0.92	0.96	0.93
History	0.97	0.95	0.96	0.96
Horror	0.89	0.89	0.89	0.89
Music	0.96	0.96	0.96	0.96
Mystery	0.95	0.92	0.95	0.92
Romance	0.84	0.79	0.84	0.80
Sci_Fi	0.94	0.91	0.94	0.92
Thriller	0.83	0.78	0.83	0.79
War	0.97	0.94	0.97	0.95
Western	0.98	0.97	0.97	0.97

1. **Description of Metrics**

   Our team chose to use prediction accuracy against the test set as the main performance metric. Real-world tasks didn't govern the analysis, so the tradeoffs offered by precision, recall, and F1 scores were not as applicable here. Similarly, the abstract classification task seemed neutral enough not to need the fine-tuning offered by ROC curves with prediction probability thresholds. For this reason, a simple accuracy metric was effective for model evaluation with this milestone. We reported both per-genre and overall accuracy.

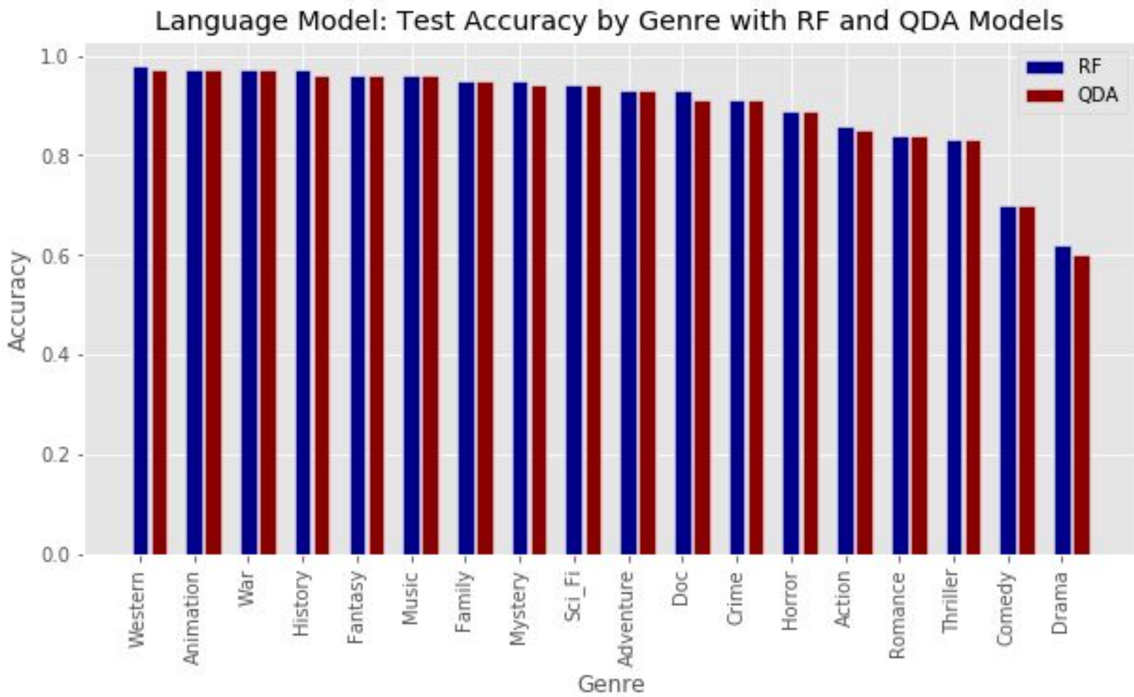
2. **Evaluation**

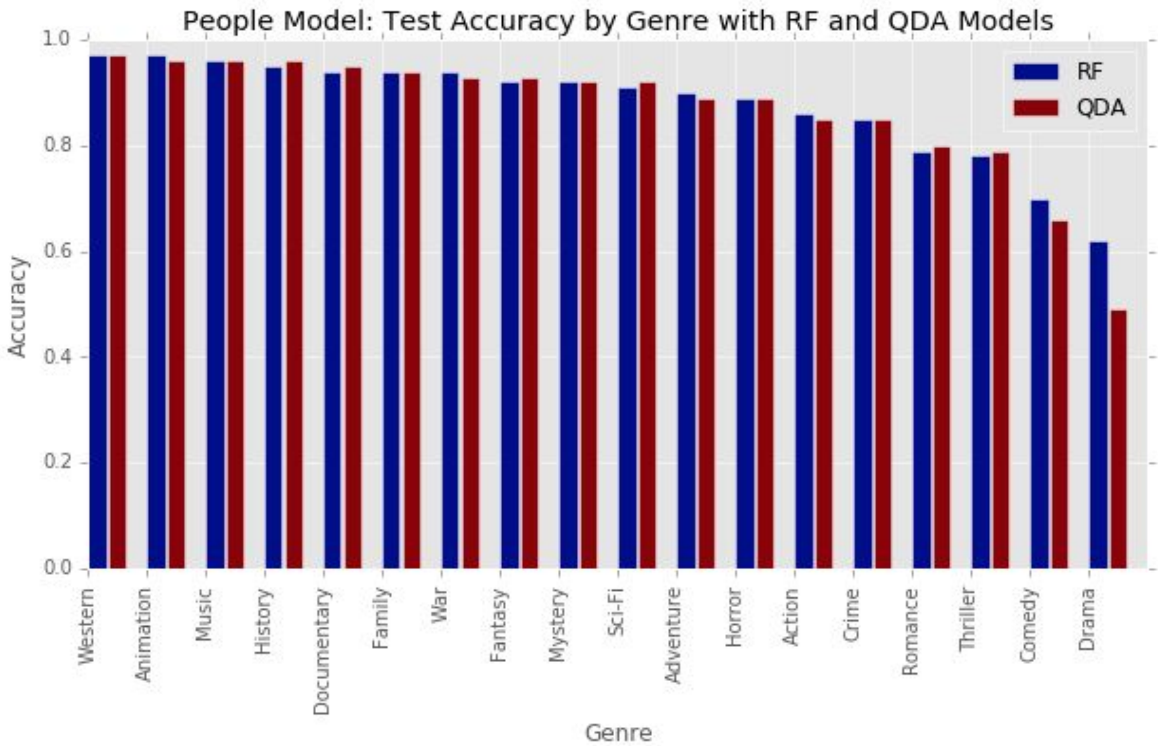
   For both algorithms and both predictor sets, we divided the data into training and testing sets (with 80% in the training set and 20% in the testing set). We trained the algorithms exclusively on the training data, while scoring exclusively on the testing data. This traditional approach should avoid any contamination of prediction evaluation.

   For the language model, it was essential to proceed carefully with the two processing steps prior to running the machine learning algorithms. When creating a document-term matrix, we fitted the vectorization only on the training data, and then applied this vectorization to the testing data when needed. Similarly when performing dimensionality reduction, we fit the principal components only on the training data, and used these components with the testing data when needed. Fortunately, the scikit-learn package provides good support for separating this processing into fitting and transformation steps.

3. Visualizations

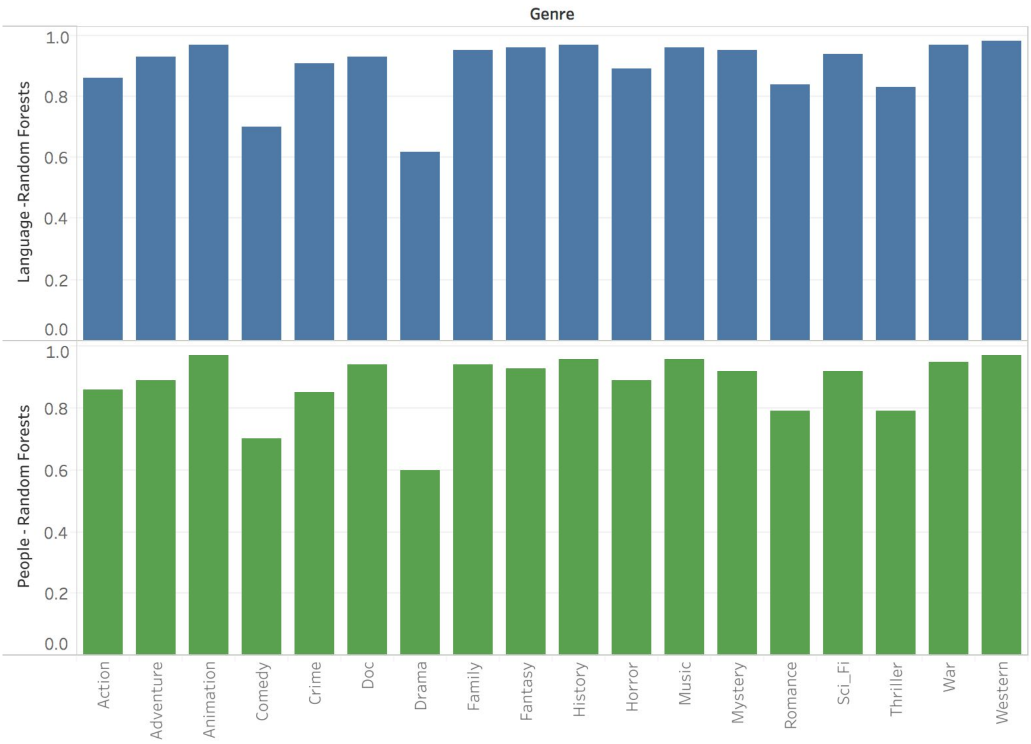
Although the graphing order of accuracy by genre is not identical between the language and people models, both models follow a similar trend. Comedy and Drama are the most challenging genres to predict across both datasets. In almost all cases, Random Forest slightly outperforms QDA.



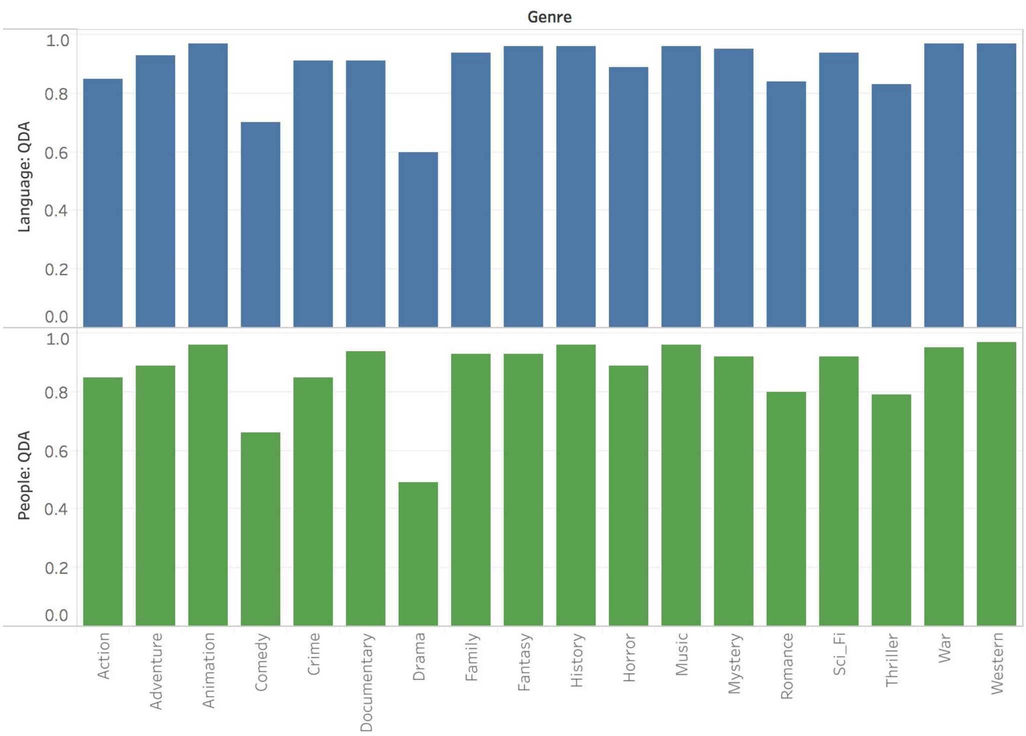


Aligning the genre graphs for all four models reaffirms our earlier observation that comedy and drama are challenging to predict with both language and people data.

Random Forests: Comparision of Models



QDA: Test Accuracy by Models



Comparison of Two Models - Strengths and Weaknesses

As seen above, performance by all four variations of our modeling are similar. Both random forests and QDA achieved comparable prediction accuracy with this data, though random forests had an edge over QDA with the people model. This similarity is explored further in the discussion section below.

One advantage of the QDA technique was computational efficiency. Processing time was 10%-20% of that required by the random forest technique, informally speaking. In a production situation, if one were firmly convinced of the reliability and performance level of QDA, preferring this technique over random forests would enable both scaling up to larger data sizes and faster turnaround of analysis deliverables.

Still, given sufficient computing resources, it appears that random forests has a small advantage over QDA in prediction accuracy with this data set. The only exception is the Documentary genre, where QDA slightly outperformed random forests. This difference was not significant, and increased sampling might diminish that gap.

Discussion of Performance and Possible Improvements

With overall accuracy between 87% and 90% the performance achieved by these models is promising. Although accuracy was not perfect, it was high enough to demonstrate signal found in the data. Notably, achieving comparable performance with two algorithms on two predictor sets indicates that limitations might exist in the structure of the domain. That is, it may be that the application of certain genre tags is inherently fuzzy, which reduces prediction accuracy with those genres. Drama and comedy exhibited the lowest accuracies in this analysis, perhaps indicating relatively loose definitions in the minds of the taggers.

The approximate equivalence of random forests and QDA was surprising. Anecdotally, random forests are a more powerful, off-the-shelf technique. So the high accuracy with random forests could somewhat be expected. Given that QDA more or less equalled this performance, we can infer that the decision boundary for each genre tag is quadratic (after applying PCA). It appears that each genre tag follows a multivariate Gaussian distribution in the reduced space defined by the PCA transformation. Interestingly, increasing the number of principal components did not materially affect the results.

A natural improvement to the models described above is to fine-tune the hyper-parameters through cross-validation. Because of processing constraints, this cross-validation might be relatively time-consuming, so deserving of more focus than our team was able to dedicate during the project schedule. A tuning grid would be appropriate to support the cross-validation -- tree number and tree depth for random forests, and regularization and priors for QDA. In addition, tuning the number of principal components may be worthwhile.

Another possible improvement is to produce a stacked model using the the movie-genre probability matrices output from each of our four models. This stacked or ensemble model could prove to be more accurate than each model individually.

Conclusion

Classic machine learning delivered satisfactory results for prediction accuracy with this data set. Nevertheless, our team encountered an apparent upper limit to the power of traditional methods with this data set. It is unclear whether this limitation is inherent in the domain - tagging fuzziness, as speculated above - or whether this limitation can be overcome with more advanced techniques. During Milestone 4, our team plans to to explore deep learning in order to resolve this issue. We will pursue both from-scratch and pre-trained neural networks, scaling up from pilot implementation on personal computers to GPU support on AWS. It will be interesting to compare the results from classic machine learning with those from deep learning.

imdb_people_model

April 19, 2017

1 CS 109B - Course Project - Team 14

1.1 Genre Prediction - IMDb - by Directors, Actors, Writers & Musicians

```
In [1]: import sqlite3
        import re

        import numpy as np
        import os.path as op
        import pandas as pd
        import math

        from matplotlib import pyplot as plt
        from sklearn.cross_validation import KFold
        from sklearn.model_selection import train_test_split as sk_split
        from sklearn.decomposition import TruncatedSVD as tSVD
        from sklearn.feature_extraction.text import CountVectorizer
        from sklearn.feature_extraction.text import TfidfVectorizer
        from scipy.io import mmwrite
        from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
        from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as
        from sklearn.ensemble import AdaBoostClassifier as Boost
        from sklearn.ensemble import RandomForestClassifier as RFC
        from sklearn.linear_model import LinearRegression as Lin_Reg
        from sklearn.linear_model import LogisticRegression as Log_Reg
        from sklearn.neighbors import KNeighborsClassifier as KNN
        from sklearn.svm import SVC

        from IPython.display import display, HTML, Markdown
        %matplotlib inline
        plt.style.use('ggplot')
        def printmd(string):
            display(Markdown(string))
```

```
/Applications/Anaconda/anaconda/lib/python2.7/site-packages/matplotlib/font_manager
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take
/Applications/Anaconda/anaconda/lib/python2.7/site-packages/sklearn/cross_validatio
```

```
  "This module will be removed in 0.20.", DeprecationWarning)
```

```
In [2]: movie_fields = ['imdb_id',
                        'Action','Adventure','Animation','Comedy','Crime','Document
                        'Family','Fantasy','History','Horror','Music','Mystery',
                        'Romance','Sci-Fi','Thriller','War','Western']

        genres = movie_fields[1:len(movie_fields)]

        num_genres = len(genres)
        num_non_genre_cols = 1

        train_percent = 80
```

1.2 import data

```
In [3]: #import clean IMDB data

        #ignore ,'Game-Show','News','Reality-TV', 'Biography','Adult','Film-Noir',
        df_im = pd.read_csv( './datasets/output/imdb_movies_trim.csv'
                            , encoding='utf-8'
                            , usecols = movie_fields)
        #reorder the columns
        df_im = df_im[movie_fields]
        #make genre names consistent with tmdb
        #df_im = df_im.rename(columns=['imdb_id'] + genres)

        #import people (ignore name)
        df_im_people = pd.read_csv('./datasets/output/imdb_people_trim.csv'
                                   ,encoding='utf-8'
                                   ,usecols = ['imdb_id','person_id','role_id'])
        df_im_people.drop_duplicates(inplace=True)
```

```
In [4]: # get overview of data
        print
        print("Data dimensions: " + str(df_im.shape))
        display(df_im.head())
```

```
Data dimensions: (36007, 19)
```

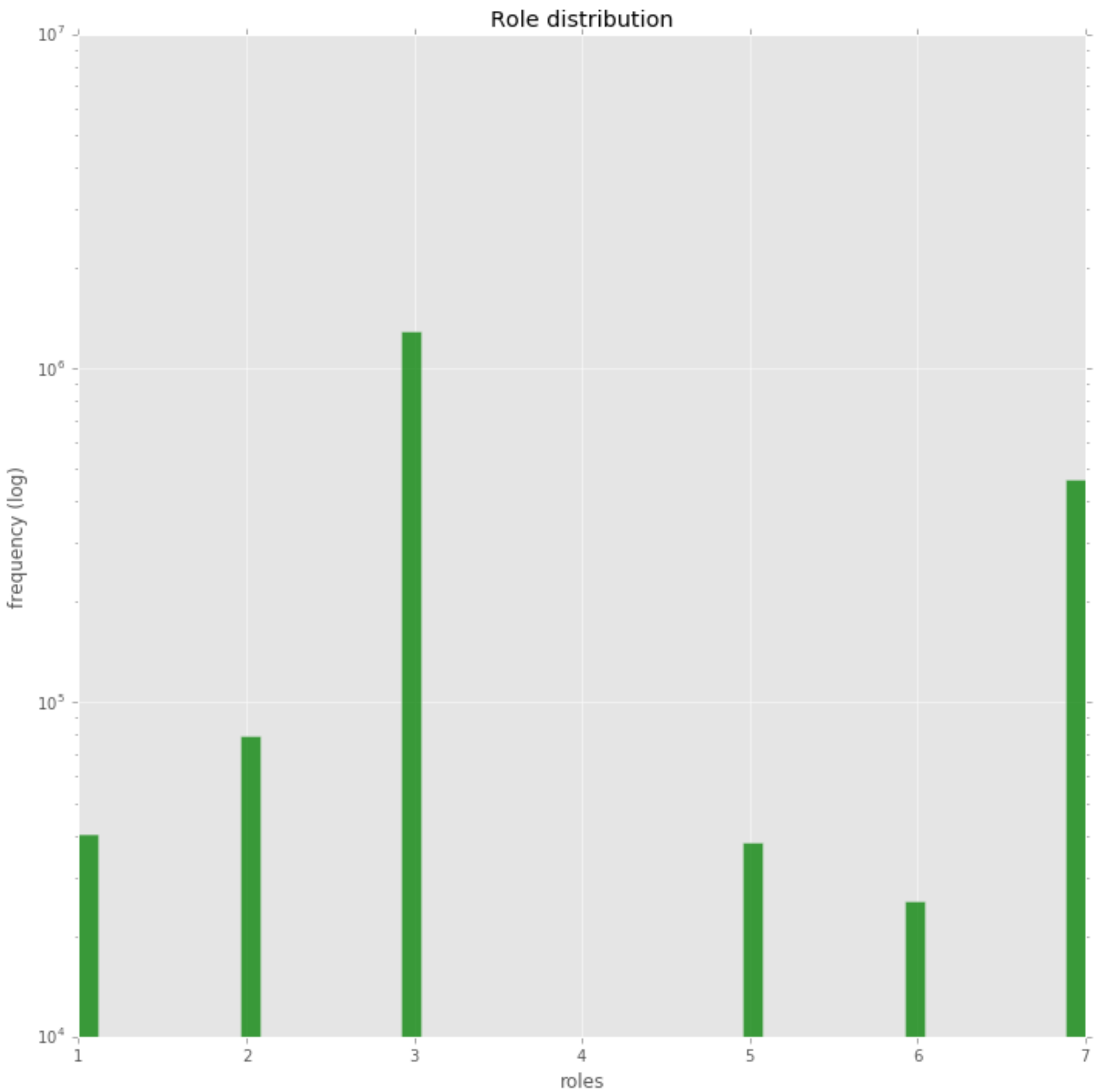
```
   imdb_id  Action  Adventure  Animation  Comedy  Crime  Documentary  Drama  \
0   114709     0.0        1.0        1.0     1.0    0.0          0.0    0.0
1   113497     1.0        1.0        0.0     0.0    0.0          0.0    0.0
2   113228     0.0        0.0        0.0     1.0    0.0          0.0    0.0
3   114885     0.0        0.0        0.0     1.0    0.0          0.0    1.0
4   113041     0.0        0.0        0.0     1.0    0.0          0.0    0.0
```

```
   Family  Fantasy  History  Horror  Music  Mystery  Romance  Sci-Fi  \
0     1.0      1.0      0.0     0.0    0.0      0.0      0.0     0.0
1     1.0      1.0      0.0     0.0    0.0      0.0      0.0     0.0
2     0.0      0.0      0.0     0.0    0.0      0.0      1.0     0.0
3     0.0      0.0      0.0     0.0    0.0      0.0      1.0     0.0
4     1.0      0.0      0.0     0.0    0.0      0.0      1.0     0.0

   Thriller  War  Western
0       0.0  0.0      0.0
1       1.0  0.0      0.0
2       0.0  0.0      0.0
3       0.0  0.0      0.0
4       0.0  0.0      0.0
```

```
In [5]: #function for plotting histograms
        def plot_hist(data, title, x_label, face, axes, log=False):

            axes.hist(   data,
                         50,
                         normed=0,
                         facecolor=face,
                         alpha=0.75,
                         log = log
                     )

            axes.set_title(title)
            axes.set_xlabel(x_label)
            str_y_label = 'frequency'
            if log:
                str_y_label = str_y_label + ' (log)'

            axes.set_ylabel(str_y_label)

            return axes
```

```
In [6]: #plot histograms for each marker and each demographics
        #in the following, instead of adding one subplot to a 4x2 grid at a time
        #I can get all the subplot axes for the grid in one line
        fig, ax1 = plt.subplots(1, 1, figsize=(10, 10))

        ax1 = plot_hist(df_im_people['role_id'].values,
                        'Role distribution',
                        'roles',
                        'green',
                        ax1, log = True)
```

```
plt.tight_layout()
plt.show()
```



For 36,000 movies we have around 2 million people/movie/role combinations. We need to get that down. Ae will start by filtering out some roles. While Actors is the bigest role we will keep it as we suspect it may be highly correlated to genre.

role_id / role

Use These Roles - 1 director - 2 writer - 3 cast - 5 original music

Ignore these Roles - 4 production-manager - 6 casting director - 7 visual effects

```
In [7]: #write movie_people_role data to in-memory SQLite db for preprocessing
        con = sqlite3.connect(":memory:")
        #con.isolation_level = None
        c = con.cursor()
```

```
df_im_people.to_sql('movie_people_role', con, index=False)
#df_im_people = None #Free up memory
```

Get unique movie, person combinations
a person may have more then one role in the movie.
Drop role and get distinct movie, person combination
Consider only director,writer,cast and original music roles

```
In [8]: c.execute('Create table movie_people as                                      ' +
                  'SELECT imdb_id, person_id                                         ' +
                  '    , count(*) as roles                                           ' +
                  '    , sum(CASE role_id WHEN 3 THEN 1 ELSE 0 END) as is_actor ' +
                  'FROM movie_people_role                                            ' +
                  'WHERE role_id in (1,2,3,5)'
                  'GROUP BY imdb_id, person_id')

        #index movie_people for better query optimisation
        c.execute("CREATE UNIQUE INDEX IF NOT EXISTS pk_movie_person on movie_peopl
        c.execute("CREATE UNIQUE INDEX IF NOT EXISTS person_movie    on movie_peopl

        #Free up memory
        c.execute("DROP TABLE movie_people_role")

Out[8]: <sqlite3.Cursor at 0x11b2748f0>
```

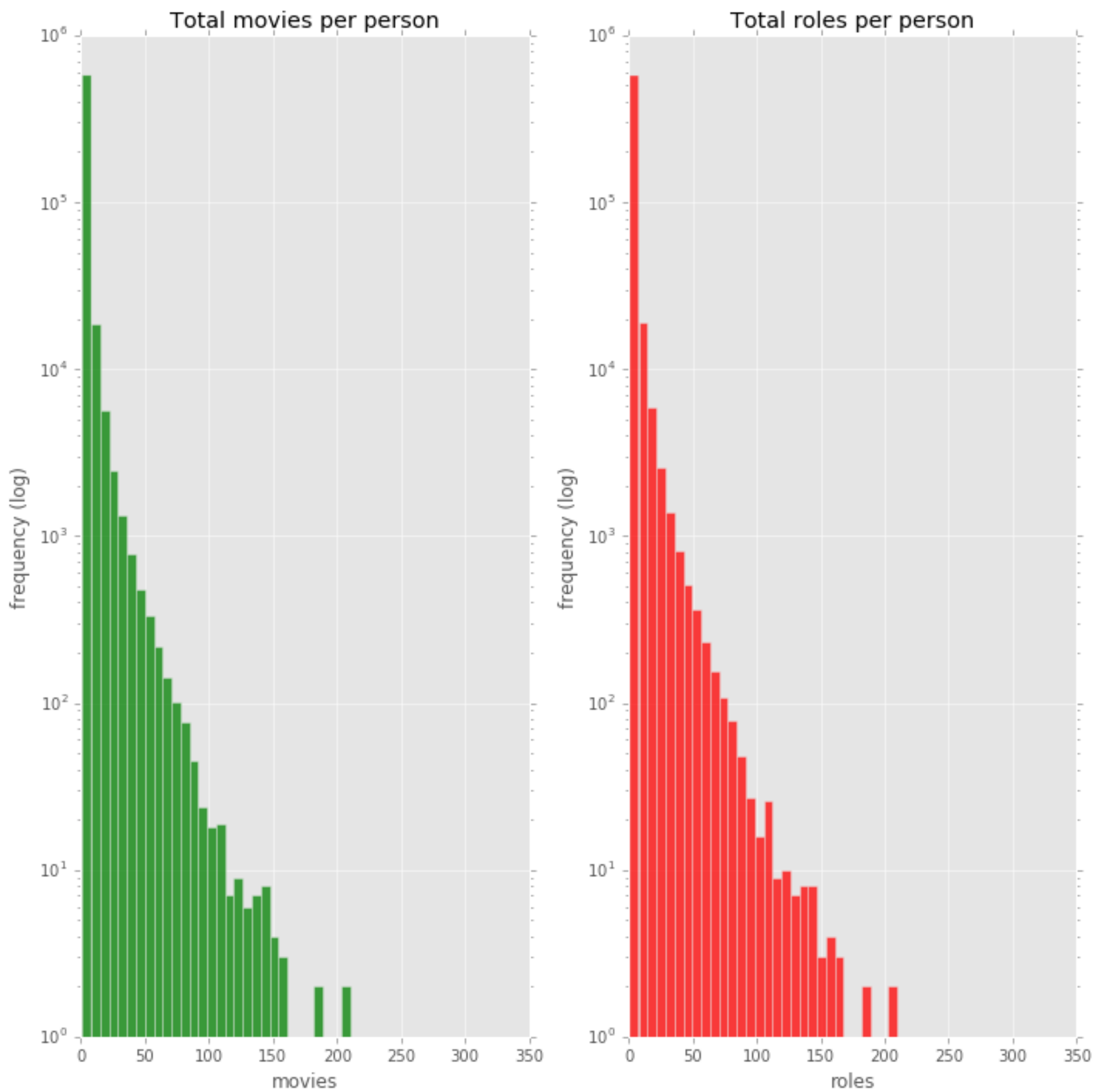
Count how many movies a person has been involved in and the number of roles

```
In [9]: c.execute('Create table people as ' + \
                  'SELECT DISTINCT person_id, count(imdb_id) as movies, sum(roles)
                  'FROM movie_people ' + \
                  'GROUP BY person_id')

        #index people for better query optimisation
        c.execute("CREATE UNIQUE INDEX IF NOT EXISTS pk_person       on people (per
        c.execute("CREATE INDEX IF NOT EXISTS person_movie_count     on people (mov

Out[9]: <sqlite3.Cursor at 0x11b2748f0>
```

Look at the total number of movies a person is involved and the number of roles

```
In [10]: df_people = pd.read_sql('select * from people', con)

         strsql =  'select   count(distinct imdb_id)   as movies    ' + \
                   '       , count(*) as people                     ' + \
                   '       , Sum(roles) as roles                    ' + \
                   'from movie_people                               '
         df_movies_count = pd.read_sql(strsql, con)
```

```
#plot histograms for each marker and each demographics
#in the following, instead of adding one subplot to a 4x2 grid at a time
#I can get all the subplot axes for the grid in one line
fig, ((ax1, ax2)) = plt.subplots(1, 2, figsize=(10, 10))

ax1 = plot_hist(df_people['movies'].values,
                'Total movies per person',
                'movies',
                'green',
                ax1,
                log = True
               )

ax2 = plot_hist(df_people['roles'].values,
                'Total roles per person',
                'roles',
                'red',
                ax2,
                log = True
               )

plt.yscale('log')
plt.tight_layout()
plt.show()

df_movies_count
```

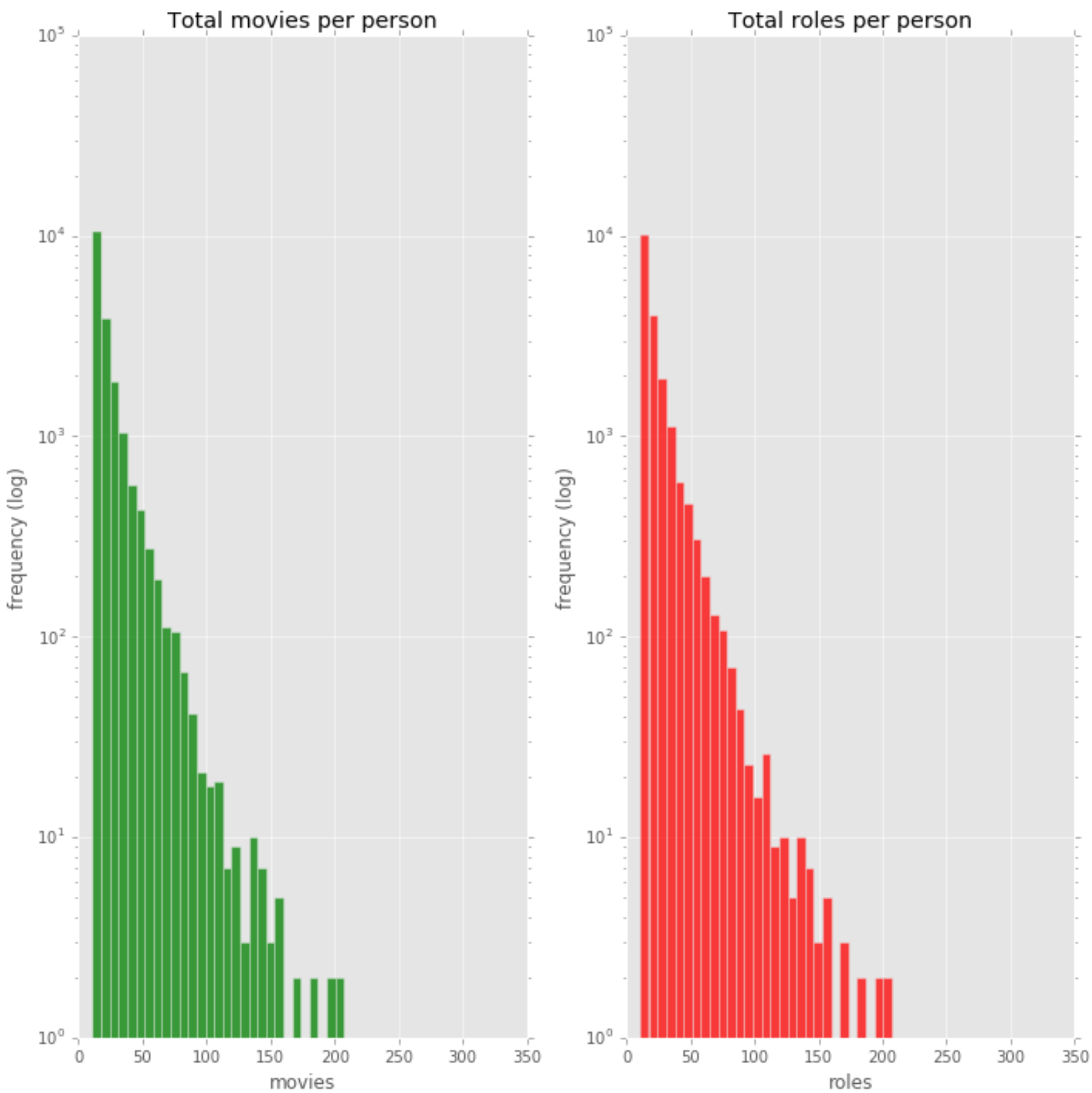



```
Out[10]:    movies   people    roles
         0   36003  1420793  1453194
```

After filtering the roles, we still have 1.4 million people. The majority of people have been involved in only one or two movies. Selecting people who have been involved more than 10 movies reduced the number of people down to 419 thousand. The trade off is that it leaves ~2,00 movies with no people asigned to it.

```
In [11]: df_people = pd.read_sql('select * from people where movies > 10', con)

         strsql =  'select count(distinct mp.imdb_id)    as movies ' + \
                   '      , count(mp.person_id)          as people ' + \
                   '      , sum(mp.roles)                as roles  ' + \
```

```
          'from people p                                ' + \
          ' inner join movie_people mp                  ' + \
          '        on (p.person_id = mp.person_id)      ' + \
          'where p.movies > 10                          '


df_movies_count = pd.read_sql(strsql, con)

#plot histograms for each marker and each demographics
#in the following, instead of adding one subplot to a 4x2 grid at a time
#I can get all the subplot axes for the grid in one line
fig, ((ax1, ax2)) = plt.subplots(1, 2, figsize=(10, 10))

ax1 = plot_hist(df_people['movies'].values,
                'Total movies per person',
                'movies',
                'green',
                ax1,
                True)

ax2 = plot_hist(df_people['roles'].values,
                'Total roles per person',
                'roles',
                'red',
                ax2,
                True)

plt.tight_layout()
plt.show()

df_movies_count
```




```
Out[11]:    movies  people   roles
         0   32017  419060  429704
```

Merge movie responses and people matrix

```
In [12]: #extract the movie/people combinateions we are interested in
         strsql =  'select mp.imdb_id     as imdb_id               ' + \
                   '     , mp.person_id   as person_id             ' + \
                   'from people p                                  ' + \
                   ' inner join movie_people mp                    ' + \
                   '        on (p.person_id = mp.person_id)        ' + \
                   'where p.movies > 10                            '
         df_people = pd.read_sql(strsql, con)
```

```
#pivot from stacked db list to 1 wide row per movie
#columns now (imdb_id, person_1, person_2,...persion_n)
df_people_xt = pd.crosstab(df_people.imdb_id, df_people.person_id, margins

#merge by movie_id to make sure the response and predictors align
df_people_xt['imdb_id_p'] = df_people_xt.index
data = pd.merge(left = df_im , right = df_people_xt, left_on='imdb_id', ri

#drop duplicate IMDB_id column
data = data.ix[:,:-2]
```

split into (x and y) and (test and train)

```
In [13]: #split into x (people) and y (genres)
         y = data.ix[:,:len(df_im.columns)]
         x = data.ix[:,len(df_im.columns):-1]

         #split into test & train
         n_samples = x.shape[0]
         train = np.random.uniform(size=n_samples) > float(train_percent) / 100.

         x_train = x[train]
         y_train = y[train]

         x_test = x[~train]
         y_test = y[~train]
```

Apply TSVD to reduce predictors

```
In [14]: # apply SVD to people matrix (train)
         tsvd = tSVD(n_components = 100)
         tsvd.fit(x_train)
         x_train_pca = tsvd.transform(x_train)
         print
         print("Cumulative percentage of variance explained: " + \
               str(round(tsvd.explained_variance_ratio_.sum(), 4)))
```

```
Cumulative percentage of variance explained: 0.1197
```

```
In [15]: pc1 = tsvd.components_[1]
         pc1_top_loadings = np.where(pc1 > 10**-1.75)

In [16]: # apply SVD to people matrix (test)
         x_test_pca = tsvd.transform(x_test)
```

1.2.1 Utility Functions

In [17]:
```
# define model codes
log_reg = 2
lda = 3
qda = 4
knn = 5
rfc = 6
boost = 7
svm = 8
```

In [18]:
```
# function to return name of model type
def get_model_name(model_type):
    if model_type == log_reg:
        model_name = "logistic regression"
    elif model_type == lda:
        model_name = "LDA"
    elif model_type == qda:
        model_name = "QDA"
    elif model_type == knn:
        model_name = "KNN"
    elif model_type == rfc:
        model_name = "random forests"
    elif model_type == boost:
        model_name = "boost"
    elif model_type == svm:
        model_name = "SVM"
    else:
        model_name = ""

    return model_name
```

In [19]:
```
# function to return unfitted model of given type
def get_model_instance(model_type, y):
    default_ratio = (y == 1).sum() / float(len(y))
    priors = (default_ratio, 1.0 - default_ratio)

    if model_type == log_reg:
         model_instance = Log_Reg(C = 1, class_weight = 'balanced',
                                  n_jobs = 3)
    elif model_type == lda:
        model_instance = LDA(priors = priors)
    elif model_type == qda:
        model_instance = QDA(reg_param = 0.25)
    elif model_type == knn:
        model_instance = KNN(n_neighbors = 5)
    elif model_type == rfc:
        model_instance = RFC(n_estimators = 20, class_weight = 'balanced',
                             max_features = 'auto', max_depth = None,
```

```
                                n_jobs = 3)
        elif model_type == boost:
            model_instance = Boost()
        elif model_type == svm:
            model_instance = SVC(kernel = 'poly', degree = 2,
                                 class_weight = 'balanced')
        else:
            model_instance = None

        return model_instance
```

```
In [20]: # function to fit and score one model of given type
         def cross_validate_one_model(x, y, model_type):
             np.random.seed(42)

             train_score_accum = 0
             test_score_accum = 0

             n_folds = 5
             kf = KFold(x.shape[0], n_folds = n_folds)
             for train_index, test_index in kf:
                 x_train, x_test = x[train_index], x[test_index]
                 y_train, y_test = y.values[train_index], y.values[test_index]

                 model = get_model_instance(model_type, y_train)
                 model.fit(x_train, y_train)
                 y_predict = model.predict(x_test)

                 train_score_accum += model.score(x_train, y_train)
                 test_score_accum += model.score(x_test, y_test)

             # calculate accuracy
             train_score = train_score_accum / float(n_folds)
             test_score = test_score_accum / float(n_folds)

             return test_score
```

```
In [21]: # function to fit and score one model of given type
         def fit_and_score_one_model(x_train, y_train, x_test, y_test, model_type):
             np.random.seed(42)

             model = get_model_instance(model_type, y_train)
             model.fit(x_train, y_train)
             y_predict = model.predict(x_test)

             train_score = model.score(x_train, y_train)
             test_score  = model.score(x_test , y_test)

             return y_predict, test_score
```

1.2.2 Modeling

Each model below is fit on training data and then scored on testing data.

Different Model Types

```
In [22]: model_types = [lda, log_reg, qda, rfc]
         model_names = ['Linear Discriminant Analysis', 'Logistic Regression', \
                        'Quadratic Discriminant Analysis', 'Random Forest Classifie
         num_model_types = len(model_types)
         test_scores_models = np.zeros(num_model_types)

         # fit and score model on each genre
         for i in range(num_model_types):
             test_scores_models[i] = \
                 cross_validate_one_model(x_train_pca, y_train.Action,
                                          model_types[i])
```

```
In [23]: test_scores_models_df = pd.DataFrame(test_scores_models, columns = ['Accur
         test_scores_models_df.Accuracy = test_scores_models_df.Accuracy.round(2)
         test_scores_models_df.index = model_names

         print
         printmd("Test Accuracy during Cross-Validation")
         display(test_scores_models_df)
```

Test Accuracy during Cross-Validation

	Accuracy
Linear Discriminant Analysis	0.18
Logistic Regression	0.78
Quadratic Discriminant Analysis	0.85
Random Forest Classifier	0.86

Different Genre Types - Random Forest

```
In [24]: y_predict_all_rf = np.zeros([y_test.shape[0], num_genres])
         test_scores_all_rf = np.zeros(num_genres)

         # fit and score model on each genre
         for i in range(num_genres):
             y_predict_all_rf[:, i], test_scores_all_rf[i] = \
                 fit_and_score_one_model(x_train_pca,
                                         y_train.iloc[:, i + num_non_genre_cols],
                                         x_test_pca,
                                         y_test.iloc[: , i + num_non_genre_cols],
                                         rfc)
```

```
In [25]: test_scores_all_rf_df = pd.DataFrame(test_scores_all_rf, columns = ['Accur
         test_scores_all_rf_df.Accuracy = test_scores_all_rf_df.Accuracy.round(2)
         test_scores_all_rf_df.index = genres

         print
         printmd("Prediction Accuracy on Testing Data (with Random Forest)")
         display(test_scores_all_rf_df)
```

Prediction Accuracy on Testing Data (with Random Forest)

```
             Accuracy
Action           0.86
Adventure        0.90
Animation        0.97
Comedy           0.70
Crime            0.85
Documentary      0.94
Drama            0.62
Family           0.94
Fantasy          0.92
History          0.95
Horror           0.89
Music            0.96
Mystery          0.92
Romance          0.79
Sci-Fi           0.91
Thriller         0.78
War              0.94
Western          0.97
```

```
In [26]: avg_score_rf = test_scores_all_rf_df.Accuracy.mean()
         printmd("")
         printmd("The average test accuracy over all genres is " + \
                 "{:.2f}".format(round(avg_score_rf, 2)))
```

The average test accuracy over all genres is 0.88

```
In [27]: printmd("")
         printmd("Description of Test Accuracy (with Random Forest)")
         test_scores_all_rf_df.describe()
```

Description of Test Accuracy (with Random Forest)

```
Out[27]:          Accuracy
         count   18.000000
```

```
mean     0.878333
std      0.097694
min      0.620000
25%      0.852500
50%      0.915000
75%      0.940000
max      0.970000
```

```
In [30]: accu_all_rf = (y_predict_all_rf ==
                        y_test.ix[:, num_non_genre_cols:(num_genres + num_non_genre_co
         per_movie_accu_rf = np.sum(accu_all_rf, axis = 1) / num_genres
         avg_per_movie_accu_rf = per_movie_accu_rf.mean()

         printmd("")
         printmd("The average accuracy of Random Forests predicting multiple genres
               "{:.2f}".format(round(avg_per_movie_accu_rf, 2)))
```

The average accuracy of Random Forests predicting multiple genres per movie is 0.88

Different Genre Types - QDA

```
In [31]: y_predict_all_qda = np.zeros([y_test.shape[0], num_genres])
         test_scores_all_qda = np.zeros(num_genres)

         # fit and score model on each genre
         for i in range(num_genres):
             y_predict_all_qda[:, i], test_scores_all_qda[i] = \
                 fit_and_score_one_model(x_train_pca,
                                         y_train.iloc[:, i + num_non_genre_cols],
                                         x_test_pca,
                                         y_test.iloc[: , i + num_non_genre_cols],
                                         qda)
```

```
In [33]: test_scores_all_qda_df = pd.DataFrame(test_scores_all_qda, columns = ['Acc
         test_scores_all_qda_df.Accuracy = test_scores_all_qda_df.Accuracy.round(2)
         test_scores_all_qda_df.index = genres

         print
         printmd("Prediction Accuracy on Testing Data (with QDA)")
         display(test_scores_all_qda_df)
```

Prediction Accuracy on Testing Data (with QDA)

```
             Accuracy
Action           0.85
Adventure        0.89
```

```
Animation        0.96
Comedy           0.66
Crime            0.85
Documentary      0.94
Drama            0.49
Family           0.93
Fantasy          0.93
History          0.96
Horror           0.89
Music            0.96
Mystery          0.92
Romance          0.80
Sci-Fi           0.92
Thriller         0.79
War              0.95
Western          0.97
```

```
In [34]: avg_score_qda = test_scores_all_qda_df.Accuracy.mean()
         printmd("")
         printmd("The average test accuracy over all genres is " + \
                 "{:.2f}".format(round(avg_score_qda, 2)))
```

The average test accuracy over all genres is 0.87

```
In [35]: printmd("")
         printmd("Description of Test Accuracy (with QDA)")
         test_scores_all_qda_df.describe()
```

Description of Test Accuracy (with QDA)

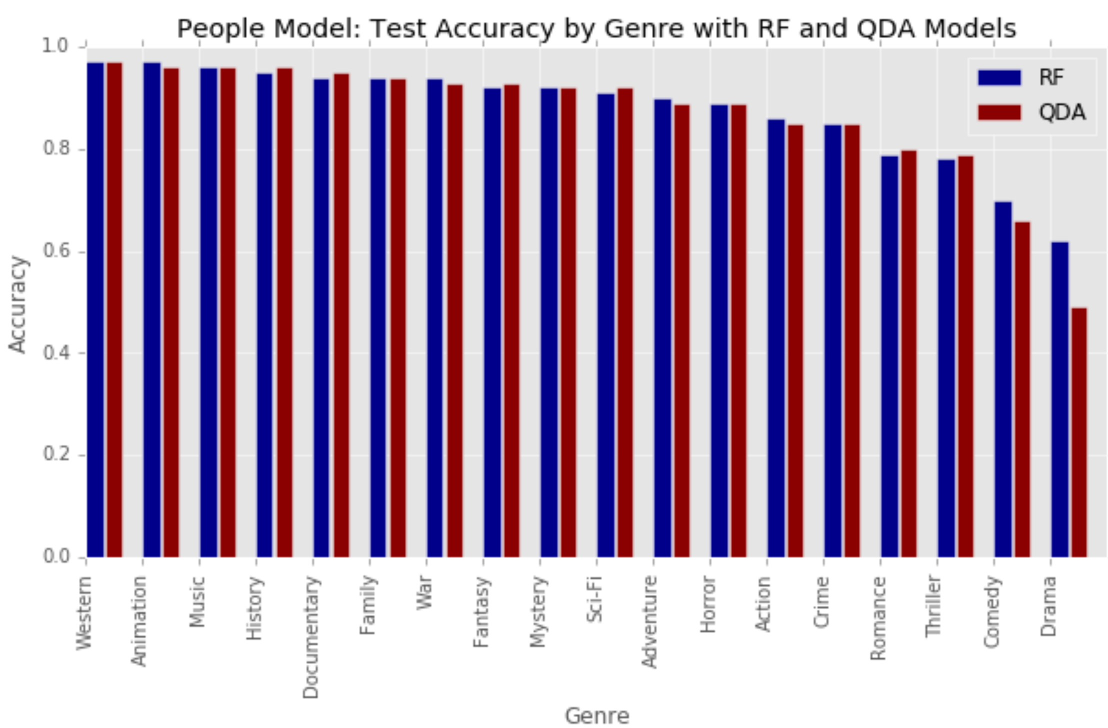
```
Out[35]:         Accuracy
         count  18.000000
         mean    0.870000
         std     0.123479
         min     0.490000
         25%     0.850000
         50%     0.920000
         75%     0.947500
         max     0.970000
```

```
In [37]: accu_all_qda = (y_predict_all_qda ==
                     y_test.ix[:, num_non_genre_cols:(num_genres + num_non_genre_co
         per_movie_accu_qda = np.sum(accu_all_qda, axis = 1) / num_genres
         avg_per_movie_accu_qda = per_movie_accu_qda.mean()

         printmd("")
         printmd("The average accuracy of predicting multiple genres per movie is '
               "{:.2f}".format(round(avg_per_movie_accu_qda, 2)))
```

The average accuracy of predicting multiple genres per movie is 0.87

Comparing Accuracy with Random Forest and QDA

```
In [39]: ### plot genre accuracy for RF vs. QDA
         test_scores_all_rf_df = test_scores_all_rf_df.sort_values(by = 'Accuracy',
         test_scores_all_qda_df = test_scores_all_qda_df.sort_values(by = 'Accuracy

         width = 0.3

         print
         fig = plt.figure(figsize = (10, 5))
         ax = fig.add_subplot(1, 1, 1)
         rects_1 = ax.bar(range(num_genres),
                test_scores_all_rf_df.Accuracy, width = width, color = 'darkblue',
                edgecolor = 'white')
         rects_2 = ax.bar(np.repeat(0.35, num_genres) + range(num_genres),
                test_scores_all_qda_df.Accuracy, width = width, color = 'darkred',
                edgecolor = 'white')
         ax.set_xticks(range(num_genres))
         ax.set_xticklabels(test_scores_all_rf_df.index.values, rotation = 90)
         ax.set_xlabel("Genre")
         ax.set_ylabel("Accuracy")
         ax.set_title("People Model: Test Accuracy by Genre with RF and QDA Models'
         ax.legend((rects_1[0], rects_2[0]), ('RF', 'QDA'))
         plt.show()
```



TMDB_Text_Mining_18Apr2017C

April 19, 2017

1 CS 109B - Course Project - Team 14

1.1 Text Mining - The Movie Database - Movie Overviews

```
In [76]: import numpy as np
         import os.path as op
         import pandas as pd

         from matplotlib import pyplot as plt
         from sklearn.cross_validation import KFold
         from sklearn.model_selection import train_test_split as sk_split
         from sklearn.decomposition import TruncatedSVD as tSVD
         from sklearn.feature_extraction.text import CountVectorizer
         from sklearn.feature_extraction.text import TfidfVectorizer
         from scipy.io import mmwrite
         from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
         from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
         from sklearn.ensemble import AdaBoostClassifier as Boost
         from sklearn.ensemble import RandomForestClassifier as RFC
         from sklearn.linear_model import LinearRegression as Lin_Reg
         from sklearn.linear_model import LogisticRegression as Log_Reg
         from sklearn.neighbors import KNeighborsClassifier as KNN
         from sklearn.svm import SVC

         from IPython.display import display, HTML, Markdown
         %matplotlib inline
         plt.style.use('ggplot')
         def printmd(string):
             display(Markdown(string))
```

```
In [2]: genres = ["Action", "Adventure", "Animation", "Comedy", "Crime", "Doc", "Drama", \
                   "Family", "Fantasy", "History", "Horror", "Music", "Mystery", \
                 "Romance", "Sci_Fi", "Thriller", "War", "Western"]

        num_genres = len(genres)
        num_non_genre_cols = 2

        train_percent = 80
```

1.2 Step 1: Load and Clean Data

In [3]:
```
# helper function to select the columns of interest from the data set
def Select_Data(data):

    features_to_select = ["overview", 'imdb_id'] + genres
    data_select = data[features_to_select]
    data_select.columns = ["Overview", "Imdb_Id"] + genres

    return data_select
```

In [4]:
```
# helper function to filter the data set down to rows of interest
def Filter_Data(data):

    # set flags for filtering
    status_flags =  ~data.Overview.isnull() & \
                    ~data.Overview.str.match('NA', na = False)

    # filter rows per flags above
    data_filter = data.ix[status_flags, :].reset_index(drop = True)

    return data_filter
```

In [5]:
```
# helper function to clean data
def Clean_Data(data):
    data_clean = data.copy()
    data_clean.Overview = data_clean.Overview.str.replace("\n|\r", ' ')

    return data_clean
```

In [6]:
```
def Preprocess_Dataset():
    clean_data_filename = "clean_tmdb_data_with_Y.csv"

    # preprocess data set and save result as new file
    if not op.isfile(clean_data_filename):
        data_raw = pd.read_csv("tmdb_data_with_Y.csv")
        data_select = Select_Data(data_raw)
        data_filter = Filter_Data(data_select)
        data_clean = Clean_Data(data_filter)
        data_clean.to_csv(clean_data_filename, index = False)

    # read pre-processed sample data file
    data_clean2 = pd.read_csv(clean_data_filename)

    return data_clean2
```

In [7]:
```
# pre-process or load data for analysis
data = Preprocess_Dataset()
```

```
# set boolean and string column data types
data.Overview = data.Overview.astype('str')

# split data into train vs. test sets
data_train, data_test = sk_split(data, train_size = train_percent / 100.0)
```

In [8]:
```
# get overview of data
print
print("Data dimensions: " + str(data_train.shape))
display(data_train.head())
```

```
Data dimensions: (26451, 20)


                                                Overview  Imdb_Id  Action  \
14562  When Martians suddenly abduct his mom, mischie...  1305591     0.0
32013  Lorsque Julien Foucault, maître d'hôtel de la ...  1314240     0.0
27122  Elmer does not want to leave Gentryville, beca...    23982     0.0
21392  Odessa is a beautiful girl addicted to the att...  1031658     0.0
17691  Double the Girls, Double the Guns!! For Mikura...   427531     1.0

       Adventure  Animation  Comedy  Crime  Doc  Drama  Family  Fantasy  \
14562        1.0        1.0     0.0    0.0  0.0    0.0     1.0      0.0
32013        0.0        0.0     1.0    0.0  0.0    0.0     0.0      0.0
27122        0.0        0.0     1.0    0.0  0.0    0.0     0.0      0.0
21392        0.0        0.0     0.0    0.0  0.0    1.0     0.0      0.0
17691        0.0        1.0     0.0    0.0  0.0    0.0     0.0      0.0

       History  Horror  Music  Mystery  Romance  Sci_Fi  Thriller  War  \
14562      0.0     0.0    0.0      0.0      0.0     0.0       0.0  0.0
32013      0.0     0.0    0.0      0.0      0.0     0.0       0.0  0.0
27122      0.0     0.0    0.0      0.0      0.0     0.0       0.0  0.0
21392      0.0     0.0    0.0      0.0      0.0     0.0       0.0  0.0
17691      0.0     0.0    0.0      0.0      0.0     0.0       1.0  0.0

       Western
14562      0.0
32013      0.0
27122      0.0
21392      0.0
17691      0.0
```

In [9]:
```
# summarize data
data_train.describe()
```

Out[9]:
```
             Imdb_Id        Action     Adventure     Animation        Comedy  \
count  2.645100e+04  26451.000000  26451.000000  26451.000000  26451.000000
```

mean	8.451764e+05	0.140222	0.063816	0.036256	0.299044
std	1.131472e+06	0.347223	0.244430	0.186929	0.457848
min	3.000000e+00	0.000000	0.000000	0.000000	0.000000
25%	7.921150e+04	0.000000	0.000000	0.000000	0.000000
50%	2.572900e+05	0.000000	0.000000	0.000000	0.000000
75%	1.356579e+06	0.000000	0.000000	0.000000	1.000000
max	6.098922e+06	1.000000	1.000000	1.000000	1.000000

	Crime	Doc	Drama	Family	Fantasy \
count	26451.000000	26451.000000	26451.000000	26451.000000	26451.000000
mean	0.087256	0.094552	0.477222	0.055385	0.041322
std	0.282215	0.292601	0.499490	0.228735	0.199037
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	1.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000

	History	Horror	Music	Mystery	Romance \
count	26451.000000	26451.000000	26451.000000	26451.000000	26451.000000
mean	0.030547	0.108918	0.034101	0.048202	0.159540
std	0.172090	0.311543	0.181492	0.214198	0.366186
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000

	Sci_Fi	Thriller	War	Western
count	26451.000000	26451.000000	26451.000000	26451.000000
mean	0.061170	0.170164	0.028997	0.025784
std	0.239646	0.375784	0.167801	0.158492
min	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	0.000000
75%	0.000000	0.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000

1.3 Step 2: Create NLP Features

In [10]:
```
# copy data for processing
data_nlp_train = data_train.copy()
data_nlp_test = data_test
```

1.3.1 Process Text

For the training data, we proceed with the following steps:

1. Stem words to reduce noise in the data

2. Fit a count vectorizer and build a document-term matrix
3. Fit principal components and reduce the data

Using a TF-IDF vectorizer, rather than a count vectorizer, gives similar prediction accuracy with this data set (not reported here). So we'll focus on the simpler counting approach for the remainder of this analysis.

```
In [11]: ### set up word stemming

         from nltk.stem import SnowballStemmer

         stemmer = SnowballStemmer(language = 'english', ignore_stopwords = True)
         analyzer = TfidfVectorizer().build_analyzer()

         def stemmed_words(doc):
             return (stemmer.stem(w) for w in analyzer(doc))

         def take(n, seq):
             seq = iter(seq)
             result = []
             try:
                 for i in range(n):
                     result.append(seq.next())
             except StopIteration:
                 pass

             return result
```

Training Data

```
In [12]: # stem words in Overview field (train)
         for index in range(data_nlp_train.shape[0]):
             data_nlp_train.Overview.values[index] = \
                 " ".join(take(1000, stemmed_words(data_nlp_train.Overview.values[index])))
```

```
In [13]: # create n-grams from overview (train)
         vectorizer = CountVectorizer(stop_words = 'english', ngram_range = (1, 1))
         vectorizer.fit(data_nlp_train.Overview.values)
         over_matrix_train = vectorizer.transform(data_nlp_train.Overview.values)
         n, p = over_matrix_train.shape
         print over_matrix_train.shape

         # term_freqs_file = "term_freqs.mtx"
         #if not op.isfile(term_freqs_file):
         #    mmwrite(term_freqs_file, over_matrix_train)
```

```
(26451, 43113)
```

```
In [14]: # apply SVD to document-term matrix (train)
         tsvd = tSVD(n_components = 100)
         tsvd.fit(over_matrix_train)
         over_matrix_pca_train = tsvd.transform(over_matrix_train)
         print
         print("Cumulative percentage of variance explained: " + \
               str(round(tsvd.explained_variance_ratio_.sum(), 4)))
```

```
Cumulative percentage of variance explained: 0.215
```

```
In [15]: pc1 = tsvd.components_[1]
         pc1_top_loadings = np.where(pc1 > 10**-1.75)
```

Testing Data Vectorizer and principal-components transformations of the testing data are performed using fits from the training data.

```
In [16]: # stem words in Overview field (test)
         for index in range(data_nlp_test.shape[0]):
             data_nlp_test.Overview.values[index] = \
                 " ".join(take(1000, stemmed_words(data_nlp_test.Overview.values[index])))

         # create n-grams from overview (test)
         over_matrix_test = vectorizer.transform(data_nlp_test.Overview.values)

         # apply SVD to document-term matrix (test)
         over_matrix_pca_test = tsvd.transform(over_matrix_test)
```

1.3.2 Explore Terms

```
In [17]: # print sample terms from overview
         feature_names = np.array(vectorizer.get_feature_names()).reshape(-1, 1)
         print "Total number of overviews and terms: " + str(n) + " overviews and " \
             + str(p) + " terms"
         print
         terms_df = pd.DataFrame(feature_names[1000:1010, 0],
                                 columns = ['Sample_Stemmed_Terms'])
         display(terms_df)
```

```
Total number of overviews and terms: 26451 overviews and 43113 terms
```

	Sample_Stemmed_Terms
0	adan
1	adana
2	adanggaman
3	adap

```
4                 adapt
5                  adar
6                   add
7                 addam
8              addendum
9               adderal
```

```
In [18]: # count words and vocabulary per overview
         data_nlp_train.Word_Count = over_matrix_train.sum(axis = 1)
         data_nlp_train.Vocab_Count = (over_matrix_train > 0).sum(axis = 1)
```

```
In [19]: # create term dictionary
         all_term_dict = zip(vectorizer.get_feature_names(),
                             np.asarray(over_matrix_train.sum(axis = 0)).ravel())
         all_term_dict_df = pd.DataFrame(all_term_dict).sort_values(by = [1],
                                                                     ascending = False)
```

```
In [20]: # list top terms in dictionary
         print
         print "Most frequent stemmed terms in overviews"
         all_term_dict_df.columns = ['Stemmed_Term', 'Count']
         all_term_dict_df.reset_index(drop = True).head(20)
```

```
Most frequent stemmed terms in overviews
```

Out[20]:

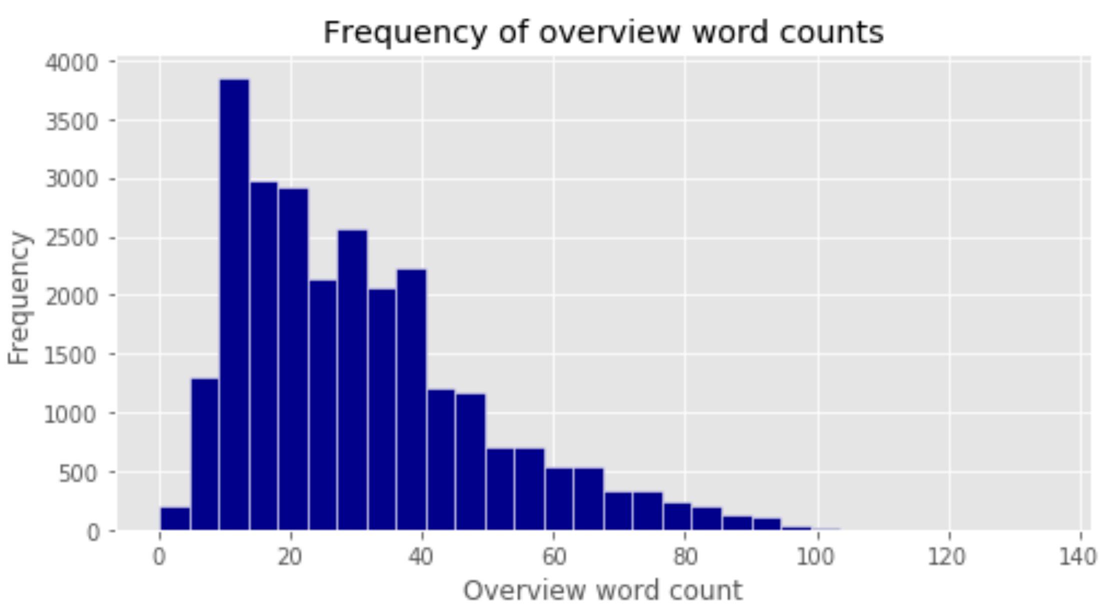
	Stemmed_Term	Count
0	life	4858
1	film	4108
2	young	3955
3	year	3870
4	love	3724
5	live	3637
6	man	3421
7	new	3367
8	stori	3206
9	famili	3164
10	friend	3029
11	world	2921
12	becom	2700
13	time	2407
14	old	2396
15	make	2262
16	father	2215
17	girl	2171
18	woman	2163
19	tri	2127

```
In [21]: terms_df = pd.DataFrame(feature_names[pc1_top_loadings], columns = ['Term'])
         printmd("#### Top loadings (terms) of principal component 1:")
         printmd(terms_df.Term.str.cat(sep = ', '))
```

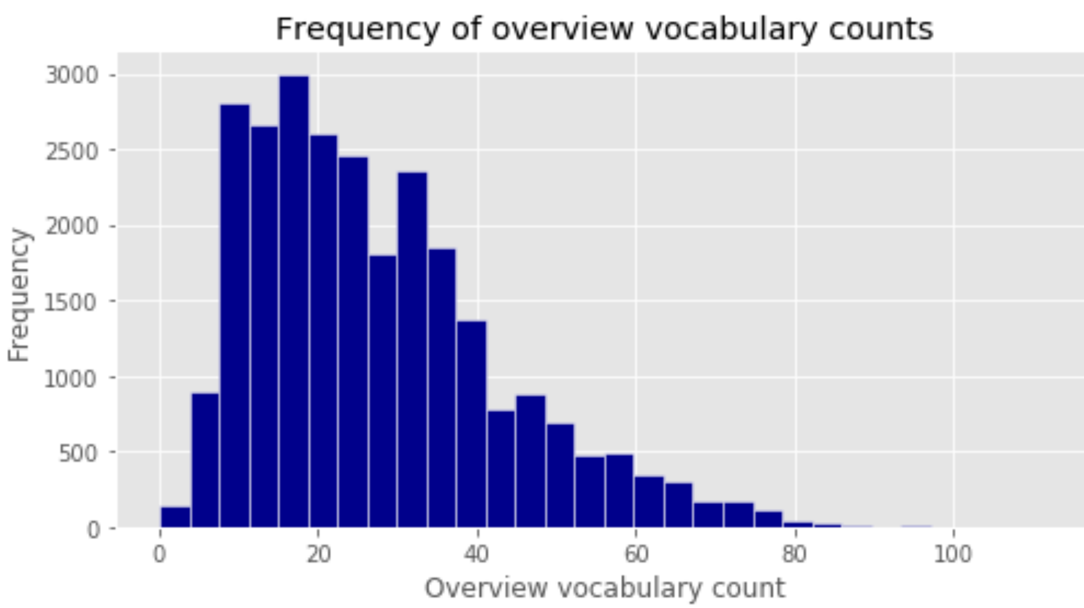
Top loadings (terms) of principal component 1: actor, adapt, american, anim, award, base, charact, cinema, comedi, creat, cultur, direct, director, documentari, drama, explor, featur, festiv, fiction, film, filmmak, follow, footag, histori, includ, interview, movi, music, narrat, novel, origin, peopl, play, produc, product, releas, role, scene, seri, set, short, shot, star, stori, tell, war, world, written

These top loadings show a variety of terms used to distinguish one genre from another. A few genre name stems appear among the top loadings, including 'anim', 'drama', 'comedi', 'documentari', 'histori', and 'music'. Other key term stems include 'war' and 'world'.

```
In [22]: # plot histogram of word counts per overview
         print
         fig = plt.figure(figsize = (8, 4))
         ax = fig.add_subplot(1, 1, 1)
         ax.hist(data_nlp_train.Word_Count, bins = 30, color = 'darkblue',
                 edgecolor = 'white')
         ax.set_title("Frequency of overview word counts")
         ax.set_xlabel("Overview word count")
         ax.set_ylabel("Frequency")
         plt.show()
```



```
In [23]: # plot histogram of vocabulary counts per overview
         print
         fig = plt.figure(figsize = (8, 4))
         ax = fig.add_subplot(1, 1, 1)
         ax.hist(data_nlp_train.Vocab_Count, bins = 30, color = 'darkblue',
                 edgecolor = 'white')
         ax.set_title("Frequency of overview vocabulary counts")
         ax.set_xlabel("Overview vocabulary count")
         ax.set_ylabel("Frequency")
         plt.show()
```



1.4 Step 3: Modeling

Accuracy is comparably good using either a random forest (RF) classifier or quadratic discriminant analysis (QDA). Let's proceed with RF, since it slightly outscores QDA by about 1%. The success of QDA implies that a quadratic decision boundary is effective for predicting individual genre with this data set. The success of QDA also implies that tagging by any single genre follows a Gaussian distribution.

Another popular machine learning algorithm, logistic regression, showed prediction accuracy about 5% less than either RF or QDA. So we'll omit logistic regression from the rest of this analysis.

Linear discriminant analysis showed prediction accuracy well below 50%, so we'll omit this method from the rest of the analysis.

Several other popular machine learning algorithms were unfortunately too slow to run effectively with the size of this data set: KNN, gradient boosting, and a support vector machine with a polynomial kernel (of degree 2).

1.4.1 Utility Functions

In [24]:
```
# define model codes
log_reg = 2
lda = 3
qda = 4
knn = 5
rfc = 6
boost = 7
svm = 8
```

In [25]:
```
# function to return name of model type
def get_model_name(model_type):
    if model_type == log_reg:
        model_name = "logistic regression"
    elif model_type == lda:
        model_name = "LDA"
    elif model_type == qda:
        model_name = "QDA"
    elif model_type == knn:
        model_name = "KNN"
    elif model_type == rfc:
        model_name = "random forests"
    elif model_type == boost:
        model_name = "boost"
    elif model_type == svm:
        model_name = "SVM"
    else:
        model_name = ""

    return model_name
```

In [26]:
```
# function to return unfitted model of given type
def get_model_instance(model_type, y):
    default_ratio = (y == 1).sum() / float(len(y))
    priors = (default_ratio, 1.0 - default_ratio)

    if model_type == log_reg:
         model_instance = Log_Reg(C = 1, class_weight = 'balanced',
                                  n_jobs = 3)
    elif model_type == lda:
        model_instance = LDA(priors = priors)
    elif model_type == qda:
        model_instance = QDA(reg_param = 0.25)
    elif model_type == knn:
        model_instance = KNN(n_neighbors = 5)
    elif model_type == rfc:
        model_instance = RFC(n_estimators = 20, class_weight = 'balanced',
                             max_features = 'auto', max_depth = None,
```

```
                                n_jobs = 3)
        elif model_type == boost:
            model_instance = Boost()
        elif model_type == svm:
            model_instance = SVC(kernel = 'poly', degree = 2,
                                 class_weight = 'balanced')
        else:
            model_instance = None

        return model_instance
```

In [27]:
```
# function to fit and score one model of given type
def cross_validate_one_model(x, y, model_type):
    np.random.seed(42)

    train_score_accum = 0
    test_score_accum = 0

    n_folds = 5
    kf = KFold(x.shape[0], n_folds = n_folds)
    for train_index, test_index in kf:
        x_train, x_test = x[train_index], x[test_index]
        y_train, y_test = y.values[train_index], y.values[test_index]

        model = get_model_instance(model_type, y_train)
        model.fit(x_train, y_train)
        y_predict = model.predict(x_test)

        train_score_accum += model.score(x_train, y_train)
        test_score_accum += model.score(x_test, y_test)

    # calculate accuracy
    train_score = train_score_accum / float(n_folds)
    test_score = test_score_accum / float(n_folds)

    return test_score
```

In [28]:
```
# function to fit and score one model of given type
def fit_and_score_one_model(x_train, y_train, x_test, y_test, model_type):
    np.random.seed(42)

    model = get_model_instance(model_type, y_train)
    model.fit(x_train, y_train)
    y_predict = model.predict(x_test)

    train_score = model.score(x_train, y_train)
    test_score = model.score(x_test, y_test)

    return y_predict, test_score
```

1.4.2 Modeling

Each model below is fit on training data and then scored on testing data.

Different Model Types

In [29]:
```
model_types = [lda, log_reg, qda, rfc]
model_names = ['Linear Discriminant Analysis', 'Logistic Regression', \
               'Quadratic Discriminant Analysis', 'Random Forest Classifier']
num_model_types = len(model_types)
test_scores_models = np.zeros(num_model_types)

# fit and score model on each genre
for i in range(num_model_types):
    test_scores_models[i] = \
        cross_validate_one_model(over_matrix_pca_train, data_nlp_train.Action,
                                 model_types[i])
```

In [30]:
```
test_scores_models_df = pd.DataFrame(test_scores_models, columns = ['Accuracy'])
test_scores_models_df.Accuracy = test_scores_models_df.Accuracy.round(2)
test_scores_models_df.index = model_names

print
printmd("Test Accuracy during Cross-Validation")
display(test_scores_models_df)
```

Test Accuracy during Cross-Validation

	Accuracy
Linear Discriminant Analysis	0.28
Logistic Regression	0.76
Quadratic Discriminant Analysis	0.86
Random Forest Classifier	0.86

Different Genre Types - Random Forest

In [31]:
```
y_predict_all_rf = np.zeros([data_nlp_test.shape[0], num_genres])
test_scores_all_rf = np.zeros(num_genres)

# fit and score model on each genre
for i in range(num_genres):
    y_predict_all_rf[:, i], test_scores_all_rf[i] = \
        fit_and_score_one_model(over_matrix_pca_train,
                                data_nlp_train.iloc[:, i + num_non_genre_cols],
                                over_matrix_pca_test,
                                data_nlp_test.iloc[:, i + num_non_genre_cols],
                                rfc)
```

```
In [32]: test_scores_all_rf_df = pd.DataFrame(test_scores_all_rf, columns = ['Accuracy'])
         test_scores_all_rf_df.Accuracy = test_scores_all_rf_df.Accuracy.round(2)
         test_scores_all_rf_df.index = genres

         print
         printmd("Prediction Accuracy on Testing Data (with Random Forest)")
         display(test_scores_all_rf_df)
```

Prediction Accuracy on Testing Data (with Random Forest)

	Accuracy
Action	0.86
Adventure	0.93
Animation	0.97
Comedy	0.70
Crime	0.91
Doc	0.93
Drama	0.62
Family	0.95
Fantasy	0.96
History	0.97
Horror	0.89
Music	0.96
Mystery	0.95
Romance	0.84
Sci_Fi	0.94
Thriller	0.83
War	0.97
Western	0.98

```
In [33]: avg_score_rf = test_scores_all_rf_df.Accuracy.mean()
         printmd("")
         printmd("The average test accuracy over all genres is " + \
                 "{:.2f}".format(round(avg_score_rf, 2)))
```

The average test accuracy over all genres is 0.90

```
In [34]: printmd("")
         printmd("Description of Test Accuracy (with Random Forest)")
         test_scores_all_rf_df.describe()
```

Description of Test Accuracy (with Random Forest)

Out[34]:

	Accuracy
count	18.000000

```
mean     0.897778
std      0.098611
min      0.620000
25%      0.867500
50%      0.935000
75%      0.960000
max      0.980000
```

```
In [35]: accu_all_rf = (y_predict_all_rf ==
                        data_nlp_test.ix[:, num_non_genre_cols:(num_genres + num_non_genre_cols)])
         per_movie_accu_rf = np.sum(accu_all_rf, axis = 1) / num_genres
         avg_per_movie_accu_rf = per_movie_accu_rf.mean()

         printmd("")
         printmd("The average accuracy of predicting multiple genres per movie is " + \
               "{:.2f}".format(round(avg_per_movie_accu_rf, 2)))
```

The average accuracy of predicting multiple genres per movie is 0.90

Different Genre Types - QDA

```
In [36]: y_predict_all_qda = np.zeros([data_nlp_test.shape[0], num_genres])
         test_scores_all_qda = np.zeros(num_genres)

         # fit and score model on each genre
         for i in range(num_genres):
             y_predict_all_qda[:, i], test_scores_all_qda[i] = \
                 fit_and_score_one_model(over_matrix_pca_train,
                                         data_nlp_train.iloc[:, i + num_non_genre_cols],
                                         over_matrix_pca_test,
                                         data_nlp_test.iloc[:, i + num_non_genre_cols],
                                         qda)
```

```
In [37]: test_scores_all_qda_df = pd.DataFrame(test_scores_all_qda, columns = ['Accuracy'])
         test_scores_all_qda_df.Accuracy = test_scores_all_qda_df.Accuracy.round(2)
         test_scores_all_qda_df.index = genres

         print
         printmd("Prediction Accuracy on Testing Data (with QDA)")
         display(test_scores_all_qda_df)
```

Prediction Accuracy on Testing Data (with QDA)

	Accuracy
Action	0.85
Adventure	0.93

```
Animation      0.97
Comedy         0.70
Crime          0.91
Doc            0.91
Drama          0.60
Family         0.94
Fantasy        0.96
History        0.96
Horror         0.89
Music          0.96
Mystery        0.95
Romance        0.84
Sci_Fi         0.94
Thriller       0.83
War            0.97
Western        0.97
```

In [38]:
```
avg_score_qda = test_scores_all_qda_df.Accuracy.mean()
printmd("")
printmd("The average test accuracy over all genres is " + \
        "{:.2f}".format(round(avg_score_qda, 2)))
```

The average test accuracy over all genres is 0.89

In [39]:
```
printmd("")
printmd("Description of Test Accuracy (with QDA)")
test_scores_all_qda_df.describe()
```

Description of Test Accuracy (with QDA)

Out[39]:

	Accuracy
count	18.000000
mean	0.893333
std	0.100762
min	0.600000
25%	0.860000
50%	0.935000
75%	0.960000
max	0.970000

In [40]:
```
accu_all_qda = (y_predict_all_qda ==
            data_nlp_test.ix[:, num_non_genre_cols:(num_genres + num_non_genre_cols)]
per_movie_accu_qda = np.sum(accu_all_qda, axis = 1) / num_genres
avg_per_movie_accu_qda = per_movie_accu_qda.mean()

printmd("")
printmd("The average accuracy of predicting multiple genres per movie is " + \
      "{:.2f}".format(round(avg_per_movie_accu_qda, 2)))
```

The average accuracy of predicting multiple genres per movie is 0.89

Comparing Accuracy with Random Forest and QDA

```
In [75]: ### plot genre accuracy for RF vs. QDA
         test_scores_all_rf_df = test_scores_all_rf_df.sort_values(by = 'Accuracy',
                                                                   ascending = False)
         test_scores_all_qda_df = test_scores_all_qda_df.sort_values(by = 'Accuracy',
                                                                     ascending = False)

         width = 0.3

         print
         fig = plt.figure(figsize = (10, 5))
         ax = fig.add_subplot(1, 1, 1)
         rects_1 = ax.bar(range(num_genres),
                test_scores_all_rf_df.Accuracy, width = width, color = 'darkblue',
                edgecolor = 'white')
         rects_2 = ax.bar(np.repeat(0.35, num_genres) + range(num_genres),
                test_scores_all_qda_df.Accuracy, width = width, color = 'darkred',
                edgecolor = 'white')
         ax.set_xticks(range(num_genres))
         ax.set_xticklabels(test_scores_all_rf_df.index.values, rotation = 90)
         ax.set_xlabel("Genre")
         ax.set_ylabel("Accuracy")
         ax.set_title("Language Model: Test Accuracy by Genre with RF and QDA Models")
         ax.legend((rects_1[0], rects_2[0]), ('RF', 'QDA'))
         plt.show()
```