CS109b - Milestone 3

Varuni Gang, Andrew Greene, Scott Shaffer, Liang-Liang Zhang

April 16, 2017

```r
library(ggplot2)
library(data.table)
library(gridExtra)
```

```
## Warning: package 'gridExtra' was built under R version 3.3.3
```

Think about how you would address the genre prediction problem with traditional statistical or machine learning methods. This includes everything you learned about modeling in this course before the deep learning part. Implement your ideas and compare different classifiers. Report your results and discuss what challenges you faced and how you overcame them. What works and what does not? If there are parts that do not work as expected, make sure to discuss briefly what you think is the cause and how you would address this if you would have more time and resources.

Detailed description and implementation of two different models

Our Approach

We decided to use a multi-label classification model (vs a multiclass classification model) approach by constructing an ensemble of single label classifiers for each of the eighteen (18) TMDb movie genres. We also decided it would be worthwhile to create more than two classificaton models for each movie genre. We can then build an ensemble model which uses the best model for each genre (based on the criteria we discuss later). Given that, it's possible that the best models for each movie genre might use different classification algorithms (Logistic Regression, Random Forest, SVM) and different sets of predictors.

Model Classes

For each movie genre, we created classification models using the following algorithms:

Dummy Model (most frequent) – Since we have highly unbalanced classes, the simplest model always predicts `0`. It will have no false positives and no true positives. This will be hard to beat in terms of classification accuracy, but other metrics such as the F1 score that consider both specificity and precision will be less forgiving.

Logistic Regression (LR) – We created several variations on Logistic Regression models for each genre. We do not expect these to be particularly effective, but they are quick to evaluate and provide a simple baseline. In particular, they give us the opportunity to assess whether predictors are effective. Again, the highly imbalanced class weights are an issue; we need to specify `class_weights='balanced'` here, and assess whether the predicted probability exceeds the base rate.

Random Forest - For this kind of classification problem, random forests are potentially quite powerful, although the tuning/training time is expensive. We allowed the tuner to run over a wide variety of hyperparameters.

Support Vector Classifier (SVC) - We used a Radial Basis Function (RBF) kernel with coarse tuning of the Cost (C) and Gamma SVM hyperparameters. Needless to say, these are also quite expensive to train.

Predictors

The following table briefly summarizes the classification models we created and which predictors were used. The predictors fall into four groups:

Base 3 TMDb data for `budget`, `runtime`, and `revenue`

Hue The most frequently occurring hue in the poster image, and the count of pixels which are that hue. (Note: Since not all poster images are the same height, the count of pixels is also influenced by the aspect ratio of the poster)

Cast-Genre Affinity The set of cast-genre affinity scores (as discussed in our Milestone 2)

The code to create various genre affinity scores from IMDB data can be found in Appendix A.

Overview Bag-of-words encoding of the 50 most frequent words in the TMDb `overview` field.

To build the bag-of-word features we extracted movie overviews (synopsis of a given movie) from TMDB data, pulled using the API during the previous milestone. We then used the Natural Language Toolkit (NTLK) package in Python to tokenize the text into separate words. We removed punctuation and stopwords using the 50 corpora and lexical resources provided by the NTLK package. We were then able to identify the 50 most common keywords present in these summaries. We transformed these keywords into predictors by assigning each a value of either `1` or `0`. A `1` designates a word present in the movie's overview, and a `0` designates a word absent from the movie's overview. The Python code can be found in Appendix B.

Model	**Feature Summary**
Most Frequent (Naive)	None
Logistic Regression	Base 3
Logistic Regression	Base 3 + Hue
Logistic Regression	Base 3 + Hue + Cast-Genre Affinity
Logistic Regression	Base 3 + Hue + Cast-Genre Affinity + Overview
Random Forest	Base 3 + Hue
Random Forest	Base 3 + Hue + Cast-Genre Affinity + Overview
SVC	Base 3 + Hue
SVC	Base 3 + Hue + Cast-Genre Affinity + Overview

Implementation

We created a Python framework (fwork.py) that allowed us to rapidly create classification models using any combination of the predictors under consideration. The framework serves as a wrapper to Python's Scikit-learn Machine Learning library and provides a standardized interface for single label classification model creation, evaluation, and comparison that can be used by the entire team.

For each class of model, we tuned the classification algorithm hyperparameters using a cross validated grid search (via Scikit-learn's GridSearchCV) on a subset of our training data (those records where `tmdb_id mod 10 == 0` or, for particularly slow models, those where `tmdb_id mod 100 == 0`), and saved the resulting "best estimator" parameters so we can return to this and do a higher-resolution grid search in that neighborhood. We then trained each model (selecting the best hyperparameters) with the full training data for this week. Finally, we computed the confusion matrix on both the training set and on this week's test set, and logged those to a `tsv` file. (We also recorded the start and end run times for each of our model runs so that we could compare clock-time performance between the models.)

We also wrote a wrapper around this Python framework that uses Spark Scikit Learn to run this cross validation grid search. This helps us to train and evaluate multiple models in parallel.

(A reminder, from previous milestones, of how we are dividing the full data set into training and test sets:

We look at the `tmdb_id` mod 10, and if it is less than 6 then it's training data; it it equals 6 then this week it is test data; and if it is greater than 6 then we are reserving it for test data in future weeks. Our intention is that each week we increment the test-set digit, thus ensuring that we don't contaminate our results by "peeking" into the future. However, we decided not to increment the test-set digit this week, since we're essentially finishing work that was begun last week, so this will let us keep one tenth of our data "in reserve".)

At this stage, we only compute and log the confusion matrices; computation of scores based on these and their analysis will follow later in this report.

The Python script that we used to evaluate different models follows:

```
''' Framework for evaluating models using Spark and Sklearn
'''

import datetime
import numpy as np
import os
import pandas
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.dummy import DummyClassifier
from sklearn.svm import SVC
if 'GridSearchCV' not in globals():
    from sklearn.grid_search import GridSearchCV as call_GridSearchCV
# To use this with Spark, write a wrapper that calls:
#   from spark_sklearn.grid_search import GridSearchCV
# and write a wrapper call_GridSearchSV to provide the Spark Context
# and then import fwork.py

from threading import Thread

# These globals will be set while loading data
imputed_values = None
train_means = None
train_sd = None
overview_feature_column_names = []

# Record model performance
# Columns are:
# Start_Time, Model_Name, Genre, Base_Rate,
# Train_TruePos, Train_FalsePos, Train_FalseNeg, Train_TrueNeg,
# Test_TruePos, Test_FalsePos, Test_FalseNeg, Test_TrueNeg,
# End_Time, GridSearch_BestParams

columns_to_scale = ['budget', 'runtime', 'revenue',
                    'H_Intensity_0', 'H_Count_0']

genre_affinity_score_column_names = [
    'score_genre_Action',
    'score_genre_Adventure',
    'score_genre_Animation',
    'score_genre_Comedy',
    'score_genre_Crime',
    'score_genre_Documentary',
    'score_genre_Drama',
    'score_genre_Family',
    'score_genre_Fantasy',
    'score_genre_Foreign',
    'score_genre_History',
    'score_genre_Horror',
    'score_genre_Music',
    'score_genre_Mystery',
    'score_genre_Romance',
    'score_genre_Science_Fiction',
    'score_genre_TV_Movie',
    'score_genre_Thriller',
    'score_genre_War',
    'score_genre_Western'
]

def load_dataset(suffix = ''):
    ''' Load all our tsv files and join them in to a
        single master dataframe
    '''
    global imputed_values
    global train_means
    global train_sd

    # Load base dataset
    dataset = pandas.read_table('data/tmdb' + suffix + '.tsv',
                                na_values = 'None',
                                dtype={
                                    'budget': float,
                                    'runtime': float,
                                    'revenue': float
                                })
    print "Basic dataset has", len(dataset), "rows"

    # Load and join simple image data
    img_hist = pandas.read_table('data/img_hist' + suffix + '.tsv',
                                  na_values = 'None',
                                  dtype={
                                      'H_Intensity_0': int,
                                      'H_Count_0': int,
                                  })
    dataset = dataset.join(img_hist[['tmdb_id',
                                      'H_Intensity_0', 'H_Count_0']],
                           on='tmdb_id',
                           rsuffix='_r')
    print "After img_hist:", len(dataset), "rows"


    # Load and join movie cast genre affinity scores
    affinity = pandas.read_table(
        'data/movie-cast-scores' + suffix + '.tsv',
        na_values = 'None',
    )
    dataset = dataset.join(affinity,
                           on='tmdb_id',
                           rsuffix='_r')

    print "After movie cast affinity:", len(dataset), "rows"

    # Load and join dircetor-genre affinity scores
    affinity_director = pandas.read_table(
        'data/movie-director-scores' + suffix + '.tsv',
        na_values = 'None',
    )
    dataset = dataset.join(affinity_director,
                           on='tmdb_id',
                           rsuffix='_director')
    print "After director affinity:", len(dataset), "rows"

    # Load and join columns from IMDB
    imdb = pandas.read_table(
        'data/imdb' + suffix + '.tsv',
        na_values = 'None')
    imdb = imdb.rename(columns= {'IMDB_ID':'imdb_id'})
    imdb['imdb_id'] = 'tt' + imdb['imdb_id'].astype(str)
    imdb = imdb.add_prefix('genre_')
    imdb = imdb.add_suffix('_imdb')
    dataset = pandas.merge(left=dataset, right = imdb,
                           left_on = "imdb_id",
                           right_on = "genre_imdb_id_imdb",
                           how='left')
    print "After IMDB:", len(dataset), "rows"

    # Load and join cast-genre affinity scores from IMDB data
    affinity_cast_imdb = pandas.read_table(
        'data/imdb-movie-cast-scores' + suffix + '.tsv',
        na_values = 'None',
    )
    affinity_cast_imdb = affinity_cast_imdb.rename(
        columns= {'IMDB_ID':'imdb_id'})
    affinity_cast_imdb['imdb_id'] = (
        'tt' + affinity_cast_imdb['imdb_id'].astype(str)
    )
    affinity_cast_imdb = affinity_cast_imdb.add_prefix('score_genre_')
    affinity_cast_imdb = affinity_cast_imdb.add_suffix('_imdb_cast')
    dataset = pandas.merge(left=dataset, right = affinity_cast_imdb,
                           left_on = "imdb_id",
                           right_on = "score_genre_imdb_id_imdb_cast",
                           how='left')
    print "After IMDB cast affinity:", len(dataset), "rows"

    # Load and join director-genre affinity scores from IMDB data
    affinity_director_imdb = pandas.read_table(
        'data/imdb-movie-director-scores' + suffix + '.tsv',
        na_values = 'None',
    )
    affinity_director_imdb = affinity_director_imdb.rename(
        columns= {'IMDB_ID':'imdb_id'})
    affinity_director_imdb['imdb_id'] = (
        'tt' + affinity_director_imdb['imdb_id'].astype(str)
    )
    affinity_director_imdb = affinity_director_imdb.add_prefix(
        'score_genre_')
    affinity_director_imdb = affinity_director_imdb.add_suffix(
        '_imdb_director')
    dataset = pandas.merge(left=dataset, right = affinity_director_imdb,
                           left_on = "imdb_id",
                           right_on = "score_genre_imdb_id_imdb_director",
                           how='left')
    print "After IMDB director affinity:", len(dataset), "rows"

    # Load and join text features based on overview
    overview_features = pandas.read_table(
        'data/overview_features' + suffix + '.tsv',
        na_values = 'None')
    overview_features = overview_features.add_prefix('overview_')
    global overview_feature_column_names
    overview_feature_column_names = list(overview_features)[1:]
    dataset = pandas.merge(left=dataset, right = overview_features,
                           left_on = "tmdb_id",
                           right_on = "overview_tmdb_id",
                           how='left')
    print "After Overview features:", len(dataset), "rows"

    # Now center and scale the data as appropriate

    # print 'Columns:', list(dataset) # column names
    print "Number of rows:", len(dataset)
    if suffix == '':
        print "Means...", datetime.datetime.now()
        if os.path.exists('data/training_means.txt'):
            train_means = pandas.Series(
                eval(open('data/training_means.txt', 'r').read()))
        else:
            train_means = dataset.mean(axis=0)
            with open('data/training_means.txt', 'w') as means_out:
                means_out.write('{')
                for (k, v) in zip(list(train_means.index), train_means):
                    means_out.write("'%s': %e,\n" % (k, v))
                means_out.write('}')
        print "SD...", datetime.datetime.now()
        train_sd = dataset.std(axis=0)
        print "SD cap...", datetime.datetime.now()
        train_sd[train_sd < 0.001] = 1  # Don't let scale blow up
        print "Computing imputed values"
        imputed_values = 0 # dataset.mean(axis=0)
        print imputed_values
    dataset = dataset.fillna(imputed_values, axis=0)
    dataset[columns_to_scale] -= train_means[columns_to_scale]
    dataset[columns_to_scale] /= train_sd[columns_to_scale]
    return dataset



print "Loading movies"
movies = load_dataset()
print "Getting small set"
small_set = movies[movies['tmdb_id'] % 10 == 0]
print "Small set Number of rows:", len(small_set)
smaller_set = movies[movies['tmdb_id'] % 100 == 0]
print "Smaller set Number of rows:", len(smaller_set)
print "Loading test set"
test = load_dataset('-test')
print "Running models"

def conf_matrix(predictions, actual):
    ''' Returns tab-delimited string of TruePos, FalsePos, FalseNeg, TrueNeg
    '''
    true_pos  = ((actual == 1) & (predictions == 1)).sum()
    false_pos = ((actual == 0) & (predictions == 1)).sum()
    false_neg = ((actual == 1) & (predictions == 0)).sum()
    true_neg  = ((actual == 0) & (predictions == 0)).sum()
    return '\t'.join([str(x) for x in
                      [true_pos, false_pos, false_neg, true_neg]])


# For a given model and selection of columns,
# - tune the model using the small subset (~2000 movies)
# - train it on all of this week's training data
# - write out the confusion matrix on both the training and test data sets
def assess_model(name,
                 model,
                 grid_params,
                 genre,
                 columns,
                 use_smaller_set = False,
                 use_base_rate_as_cutoff = False
                ):
    start_time = datetime.datetime.now()

    my_train_set = smaller_set if use_smaller_set else small_set

    try:
        # tune on a small set of the data
        gs = call_GridSearchCV(model, param_grid=grid_params)
        gs.fit(my_train_set[columns],
               my_train_set['genre_' + genre])
        # then build model on all the training data
        model = gs.best_estimator_
        model.fit(movies[columns],
                  movies['genre_' + genre])

    except Exception as e:
            print name, genre, str(e)
            with open('data/model_' + genre + '_err.tsv', 'a') as logfile:
                logfile.write(name + str(e))
            return

    train_set = movies

    if use_base_rate_as_cutoff:
        base_rate = train_set['genre_' + genre].mean()
        train_pred = model.predict_proba(train_set[columns])[:,1] > base_rate
        test_pred = model.predict_proba(test[columns])[:,1] > base_rate
    else:
        train_pred = model.predict(train_set[columns])
        test_pred = model.predict(test[columns])

    train_score = model.score(
        train_set[columns],
        train_set['genre_' + genre])
    test_score = model.score(
        test[columns],
        test['genre_' + genre])

    print name + ' ' + genre + " Train: %.3f  Test: %.3f  %d or %d test rows" % (
        train_score, test_score, len(test['genre_' + genre]), len(test_pred))

    with open('data/model_' + genre + '_log.tsv', 'a') as logfile:
        logfile.write('\t'.join([str(x) for x in [
            start_time,
            name,
            genre,
            0, # was base_rate -- now we have Dummy model for that
            conf_matrix(train_pred, train_set['genre_' + genre]),
            conf_matrix(test_pred, test['genre_' + genre]),
            datetime.datetime.now(), # endtime
            str(gs.best_params_)
        ]]) + '\n')


def assess_genre(genre):
    assess_model('Most Frequent',
                 DummyClassifier('most_frequent'),
                 {},
                 genre, ['budget', 'runtime', 'revenue'])
    assess_model('LogReg 3 predictors',
                 LogisticRegression(),
                 {'random_state': [1729,],
                  'penalty': ['l1', 'l2'],
                  'C': [.01, .1, 1, 10, 100],
                  },
                 genre, ['budget', 'runtime', 'revenue'])
    assess_model('LogReg unbalanced w/3 predictors',
                 LogisticRegression(),
                 {'random_state': [1729,],
                'penalty': ['l1', 'l2'],
                'C': [.01, .1, 1, 10, 100],
               },
              genre, ['budget', 'runtime', 'revenue'])
assess_model('LogReg unbalanced w/3 predictors and base_rate_cutoff',
              LogisticRegression(),
              {'random_state': [1729,],
               'penalty': ['l1', 'l2'],
               'C': [.01, .1, 1, 10, 100],
              },
              genre, ['budget', 'runtime', 'revenue'],
              use_base_rate_as_cutoff = True,
)
assess_model('LogReg w/H predictors',
              LogisticRegression(),
              {'random_state': [1729,],
               'penalty': ['l1', 'l2'],
               'C': [.01, .1, 1, 10, 100],
              },
              genre, ['budget', 'runtime', 'revenue',
                      'H_Intensity_0', 'H_Count_0'])
assess_model('LogReg unbalanced w/H predictors',
              LogisticRegression(),
              {'random_state': [1729,],
               'penalty': ['l1', 'l2'],
               'C': [.01, .1, 1, 10, 100],
              },
              genre, ['budget', 'runtime', 'revenue',
                      'H_Intensity_0', 'H_Count_0'])
assess_model('LogReg unbalanced w/H predictors and base_rate_cutoff',
              LogisticRegression(),
              {'random_state': [1729,],
               'penalty': ['l1', 'l2'],
               'C': [.01, .1, 1, 10, 100],
              },
              genre, ['budget', 'runtime', 'revenue',
                      'H_Intensity_0', 'H_Count_0'],
              use_base_rate_as_cutoff = True
)
assess_model('LogReg w/cast-score predictors',
              LogisticRegression(),
              {'random_state': [1729,],
               'class_weight': ['balanced', None],
               'penalty': ['l1', 'l2'],
               'C': [.01, .1, 1, 10, 100],
              },
              genre, ['budget', 'runtime', 'revenue',
                      'H_Intensity_0', 'H_Count_0',
                      'score_genre_Action',
                      'score_genre_Adventure',
                      'score_genre_Animation',
                      'score_genre_Comedy',
                      'score_genre_Crime',
                      'score_genre_Documentary',
                      'score_genre_Drama',
                      'score_genre_Family',
                      'score_genre_Fantasy',
                      'score_genre_Foreign',
                      'score_genre_History',
                      'score_genre_Horror',
                      'score_genre_Music',
                      'score_genre_Mystery',
                      'score_genre_Romance',
                      'score_genre_Science_Fiction',
                      'score_genre_TV_Movie',
                      'score_genre_Thriller',
                      'score_genre_War',
                      'score_genre_Western'
              ])
 assess_model('LogReg w/cast-score and overview_features',
              LogisticRegression(),
              {'random_state': [1729,],
               'class_weight': ['balanced', None],
               'penalty': ['l1', 'l2'],
               'C': [.01, .1, 1, 10, 100],
               },
              genre, ['budget', 'runtime', 'revenue',
                      'H_Intensity_0', 'H_Count_0'] +
              genre_affinity_score_column_names +
              overview_feature_column_names)
 # for these next ones, even the small set is too large to run
 # in a reasonable amount of time!
 assess_model('RandomForest w/H',
              RandomForestClassifier(),
              {"max_depth": [3, 4, None],
               'random_state': [1729,],
               # 'class_weight': ['balanced', None],
               'max_features': [1, 3, 5],
               # "max_features": [1, 3, 10],
               # "min_samples_split": [1, 3, 10],
               # "min_samples_leaf": [1, 3, 10],
               # "bootstrap": [True, False],
               "criterion": ["gini", "entropy"],
               # "n_estimators": [10, 20, 40, 80]}
               "n_estimators": [40, 80, 120]},
              genre,
              ['budget', 'runtime', 'revenue',
               'H_Intensity_0', 'H_Count_0'],
              use_smaller_set = True
 )
 assess_model('SVC w/H',
              SVC(),
              {'C': [0.01, 1, 100],
               'gamma': [.1, 1, 10],
               'class_weight': ['balanced',],
               'kernel': ['rbf',],
                },
               genre,
               ['budget', 'runtime', 'revenue',
                'H_Intensity_0', 'H_Count_0'],
               use_smaller_set = True
 )
 assess_model('RandomForest',
               RandomForestClassifier(),
               {"max_depth": [3, 4, None],
                'random_state': [1729,],
                # 'class_weight': ['balanced', None],
                'max_features': [1, 3, 5],
                # "max_features": [1, 3, 10],
                # "min_samples_split": [1, 3, 10],
                # "min_samples_leaf": [1, 3, 10],
                # "bootstrap": [True, False],
                "criterion": ["gini", "entropy"],
                # "n_estimators": [10, 20, 40, 80]}
                "n_estimators": [40, 80, 120]},
               genre,
               ['budget', 'runtime', 'revenue',
                'H_Intensity_0', 'H_Count_0'] +
               genre_affinity_score_column_names +
               overview_feature_column_names,
               use_smaller_set = True
 )
 assess_model('SVC',
               SVC(),
               {'C': [0.01, 1, 100],
                'gamma': [.1, 1, 10],
                'class_weight': ['balanced',],
                'kernel': ['rbf',],
                },
               genre,
               ['budget', 'runtime', 'revenue',
                'H_Intensity_0', 'H_Count_0'] +
               genre_affinity_score_column_names +
               overview_feature_column_names,
               use_smaller_set = True
 )



# Now iterate over each genre, and for each one, assess each of our models
for genre in [ "Action", "Adventure", "Animation", "Comedy", "Crime",
            "Documentary", "Drama", "Family", "Fantasy", "Foreign",
            "History", "Horror", "Music", "Mystery", "Romance",
            "Science_Fiction", "TV_Movie", "Thriller", "War", "Western", ]:
 Thread(target=assess_genre, name=genre, args=(genre,)).start()
```

Each model's log file was written separately; this simplifies things in light of the fact that our script uses multiple threads to take advantage of multiple cores. The log files are incremental, so we can do analyses while the models are running. The individual log files were then combined with:

```
o = open('data/model_@_performance.tsv', 'w')
o.write('\t'.join(['Start_Time', 'Model_Name', 'Genre',
                   'Train_TruePos', 'Train_FalsePos', 'Train_FalseNeg', 'Train_TrueNeg',
                   'Test_TruePos', 'Test_FalsePos', 'Test_FalseNeg', 'Test_TrueNeg',
                   'End_Time', 'Tuned_Params']) + '\n')

for genre in [ "Action", "Adventure", "Animation", "Comedy", "Crime",
               "Documentary", "Drama", "Family", "Fantasy", "Foreign",
               "History", "Horror", "Music", "Mystery", "Romance",
               "Science_Fiction", "TV_Movie", "Thriller", "War", "Western", ]:
    # read in accumulated performance, keeping only the last one
    rows = {}
    with open('data/model_' + genre + '_log.tsv', 'r') as i:
        for line in i:
            cols = line[:-1].split('\t')
            # work around bug in fwork if the conf matrix is 1x1
            if len(cols) < 13:
                continue
            # patch older rows that didn't capture the metadata
            if len(cols) == 13:
                cols += ['{}']
            # drop the base rate which we no longer capture
            # because it can be computed downstream
            cols[3:4] = []
            # capture the name, which is our key
            name = cols[1]
            # reconstruct the line and store it
            rows[name] = '\t'.join(cols)
    for line in rows.values():
        o.write(line + '\n')
```

This produces a tsv file which can be brought into R for visualizations

```
# mperf is the "model performance" data frame:
mperf <- read.table('data/model_@_performance.tsv',
                    header=TRUE, sep='\t')
```

For the remainder of this report, we will only be focusing on the test scores. The training scores are essentially the same.

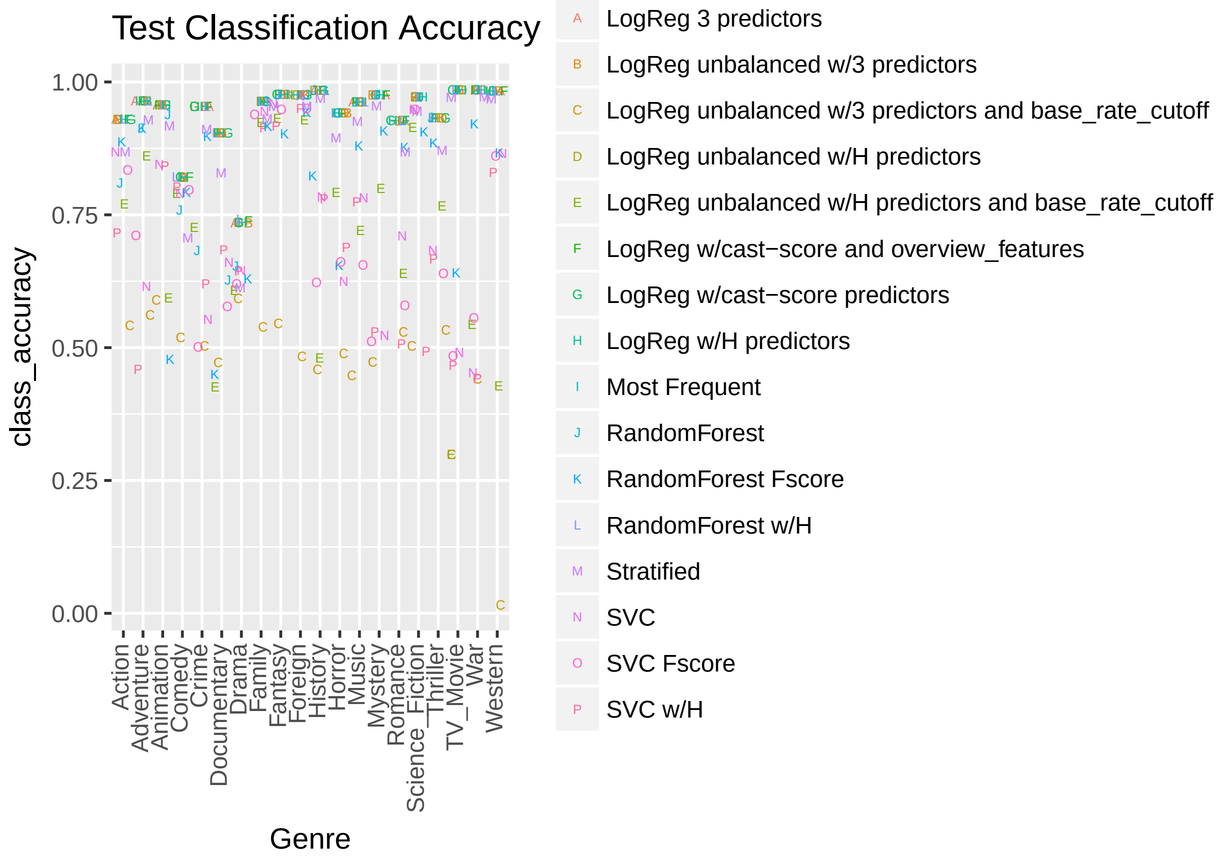
```
# Some numbers that can be computed from the confusion matrix
mperf$Test_GroundPos <-
  mperf$Test_TruePos + mperf$Test_FalseNeg
mperf$Test_GroundNeg <-
  mperf$Test_TrueNeg + mperf$Test_FalsePos
mperf$Test_NumObservations <-
  mperf$Test_GroundPos + mperf$Test_GroundNeg

mperf$Test_BaseRate <- mperf$Test_GroundPos /
  mperf$Test_NumObservations

mperf$Test_Precision <-
  (mperf$Test_TruePos) / (mperf$Test_TruePos + mperf$Test_FalsePos + 1e-10)
```

```
mperf$Test_Recall <-
  (mperf$Test_TruePos) / (mperf$Test_TruePos + mperf$Test_FalseNeg + 1e-10)
```

Visualizations

Here we compute and visualize several metrics for assessing the quality of our models

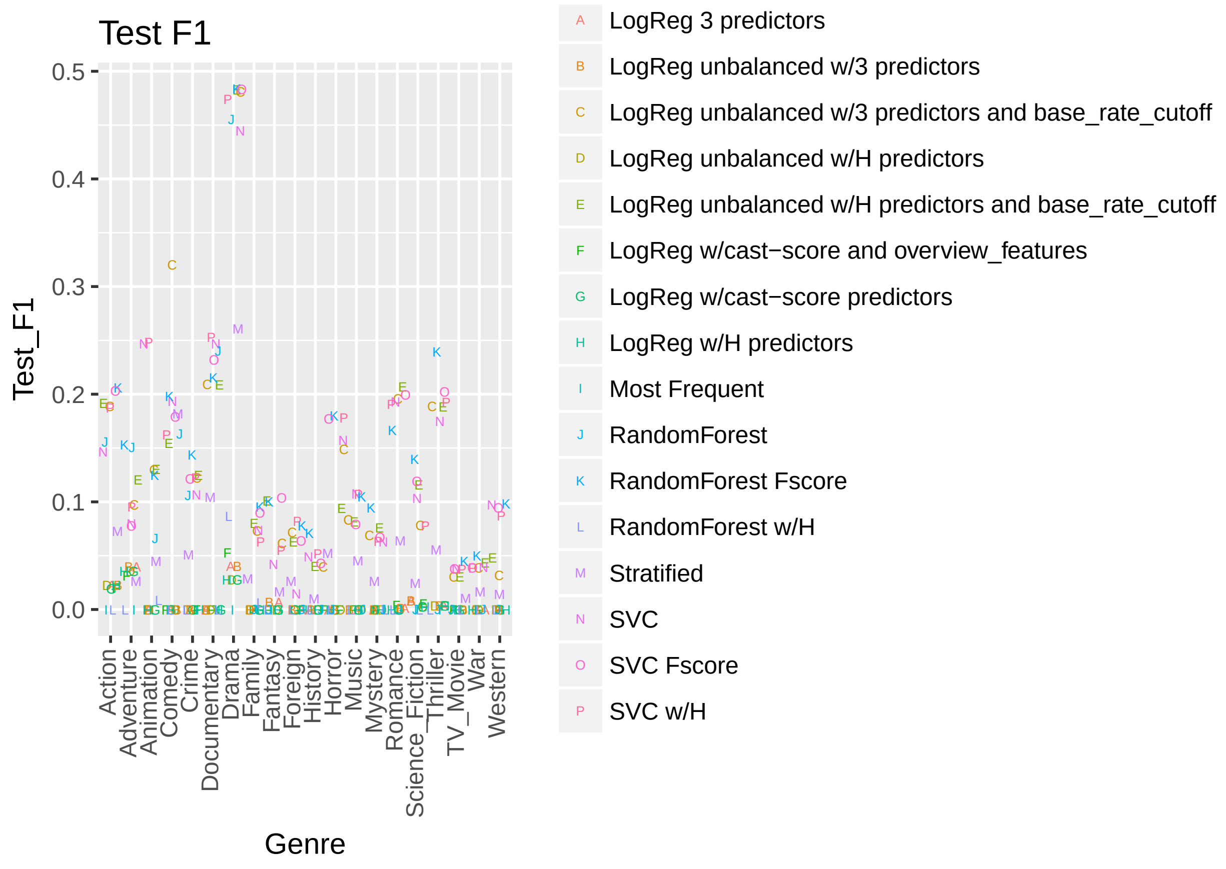
```
mperf$class_accuracy <-
  (mperf$Test_TruePos + mperf$Test_TrueNeg) /
  mperf$Test_NumObservations

ggplot(mperf,
       aes(x=Genre,
           y=class_accuracy,
           shape=Model_Name,
           color=Model_Name
           )) +
  geom_jitter(height=0) +
  scale_shape_manual(values=LETTERS) +
  labs(title='Test Classification Accuracy') +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust= 0.3))
```



As expected, classification accuracy is a poor guide because the cohorts are so imbalanced. In particlar, we see many models clustered around the base rate for the “0” outcome. (We use horizontal jittering to make the labels legible in this and the following graphs.)

We do note, however, that for almost every genre, model "E" (unbalanced with the hue-based predictors) notably outperforms model "C" (unbalanced with only the three core predictors). This strongly suggests that the two additional predictors have value. (These underperform the ones that always predict a "0" outcome, however.)

```
mperf$Test_F1 <-
  2 * (mperf$Test_Precision * mperf$Test_Recall) /
      (mperf$Test_Precision + mperf$Test_Recall + 1e-10)

ggplot(mperf,
       aes(x=Genre,
           y=Test_F1,
           shape=Model_Name,
           color=Model_Name
           )) +
  geom_jitter(height=0) +
  scale_shape_manual(values=LETTERS) +
  labs(title='Test F1') +
  theme(axis.text.x = element_text(angle = 90, hjust = 1, vjust= 0.3))
```



The F1 score has the advantage that it recognizes the tradeoff between specificity and selectivity. (See below for a more in-depth discussion.) By this measure, models "C" and "E" appear roughly equivalent. We do note that Drama is an unusual category in this plot, with all models doing better here than in other genres. Looking back at the classification accuracy plot, however, this would appear to be because the base rate is 75%, so the classes are far less imbalanced.

Assessment of metrics

Description of your performance metrics

We considered three metrics for assessing the performance of our models on each genre.

Classification Accuracy is not a useful performance metric for this particular classification problem because of the imbalanced nature of our data; all models will achieve very good accuracy, including the Naive classifier that predicts all 0s.

F1 Score is probably the most meaningful performance metric, in that it's the harmonic mean of precision and recall. This reflects the tradeoff between these two, especially in the case of imbalanaced populations: the lack of false positives does not excuse the surfeit of false negatives.

$$F_1 = \frac{2(Precision)(Recall)}{(Precision + Recall)}$$

We considered use *R ˆ2*, the coefficient of determination, but the results did not add much insight because we did not have a rich enough set of models to compare.

Because we've chosen to model each label separately, instead of building a single multi-label model or a multiclass model, we are able to choose different models for each genre. In particular, for each genre label we choose the model with the best F-score.

We can compare these ensemble models using the "Hamming Loss", which is commonly used for this purpose. The Hamming Loss is the fraction of mislabeling:

$$L_H(Ensemble) = \frac{\sum\limits_{g \in Genres} FP_g + FN_g}{(N_{Genres})(N_{Observations})}$$

where:

$\mathbf{FP_g}$ is the False Positives for specified Genre (g)

$\mathbf{FN_g}$ is the False Negatives for specified Genre (g)

$\mathbf{N_{Genres}}$ is the number of genres

$\mathbf{N_{Observations}}$ is the number of observations

As with our model, this considers each observation-genre pair to be a separate candidate for assessment. However, because it is based on 0-1 loss for each label, it is also thrown off by the highly imbalanced nature of our data. We could also consider the Macro-F1 ensemble score, which applies the F1 score approach to a multi-label environment, see https://arxiv.org/pdf/1609.00288.pdf

We can compute the Hamming loss for each of our model classes, as well as for the ensemble model. Since each model is operating over all observations for each genre, the Hamming Loss is simply 1 minus the mean of the classification accuracy for each of the genres for the given model. For the ensemble, it's 1 minus the mean of the classifcation accuracies corresponding to the highest F1 score for each genre.

These calculations are performed below.

Careful performance evaluations for both models with *Visualizations of the metrics for performance evaluation*

For each genre, logistic regression, SVM and Random Forest were trained on the training dataset. Then classification accuracy, sensitivity, specificity, precision and F1 score were evaluated for each model, using both the training and test data.

As mentioned in Milestone 2, we have noticed the imbalanced nature of the data. For all genres, populations consist of a large number of negative examples (class 0) and a small number of positive examples (class 1).

Finally, we built two ensemble models which uses the best model for each genre; once based on classification accuracy and once based on f1 score).

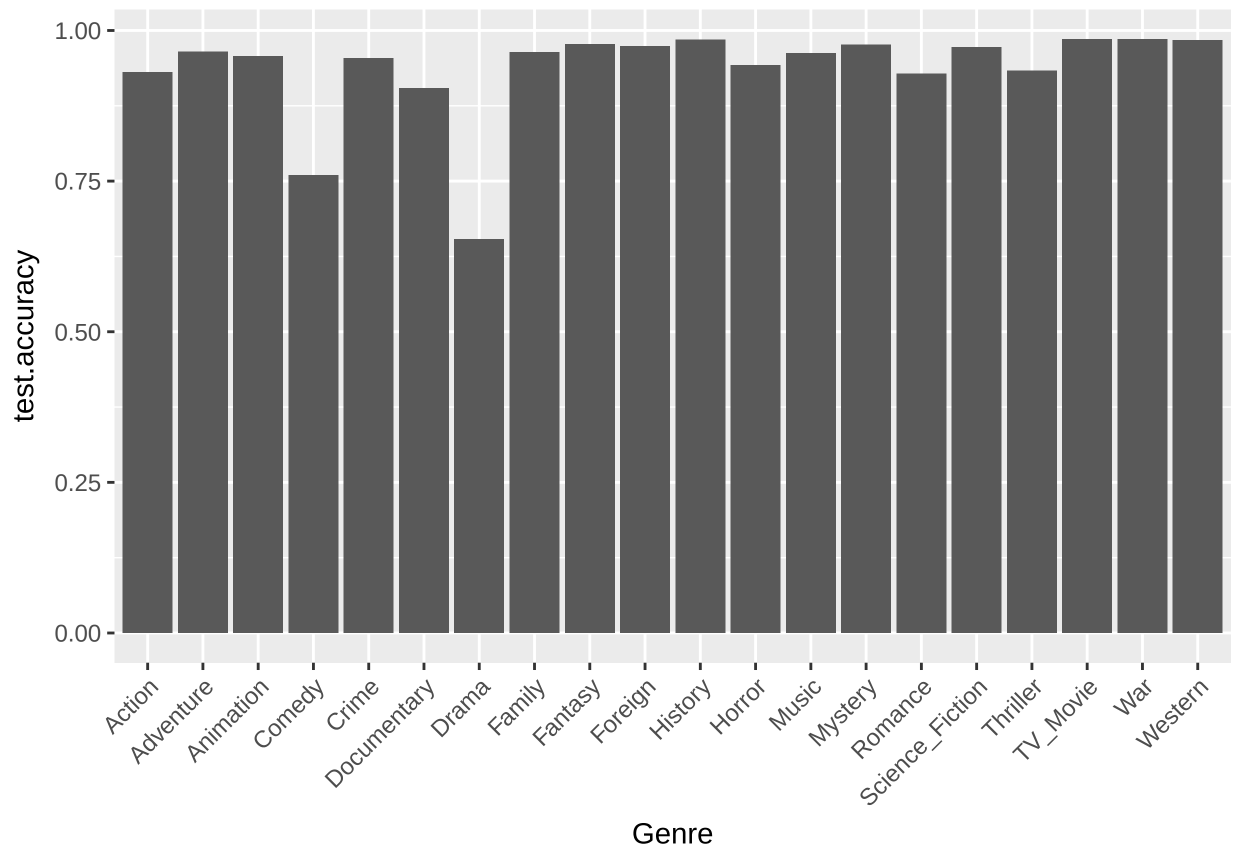
train

```r
#train
# accuracy
mperf$train.accuracy <- (mperf$Train_TruePos + mperf$Train_TrueNeg)/
  (mperf$Train_TruePos + mperf$Train_FalseNeg + mperf$Train_TrueNeg +
     mperf$Train_FalsePos)

# sensitivity
mperf$train.sensitivity <- mperf$Train_TruePos/
  (mperf$Train_TruePos + mperf$Train_FalseNeg)

# specificity
mperf$train.specificity <- mperf$Train_TrueNeg/
  (mperf$Train_FalsePos + mperf$Train_TrueNeg)

# precision
mperf$train.precision <- mperf$Train_TruePos/
  (mperf$Train_TruePos + mperf$Train_FalsePos)

# fscore:harmonic mean between precision and recall
mperf$train.fscore <- 2 * mperf$Train_TruePos /
  (2* mperf$Train_TruePos + mperf$Train_FalsePos + mperf$Train_FalseNeg)

#test
# accuracy
mperf$test.accuracy <- (mperf$Test_TruePos + mperf$Test_TrueNeg)/
  (mperf$Test_TruePos + mperf$Test_FalseNeg + mperf$Test_TrueNeg +
     mperf$Test_FalsePos)

# sensitivity
mperf$test.sensitivity <- mperf$Test_TruePos/
  (mperf$Test_TruePos + mperf$Test_FalseNeg)

# specificity
mperf$test.specificity <- mperf$Test_TrueNeg/
  (mperf$Test_FalsePos + mperf$Test_TrueNeg)

# precision
mperf$test.precision <- mperf$Test_TruePos/
  (mperf$Test_TruePos + mperf$Test_FalsePos)

# fscore:harmonic mean between precision and recall
mperf$test.fscore <- 2 * mperf$Test_TruePos /
  (2* mperf$Test_TruePos + mperf$Test_FalsePos + mperf$Test_FalseNeg)
```

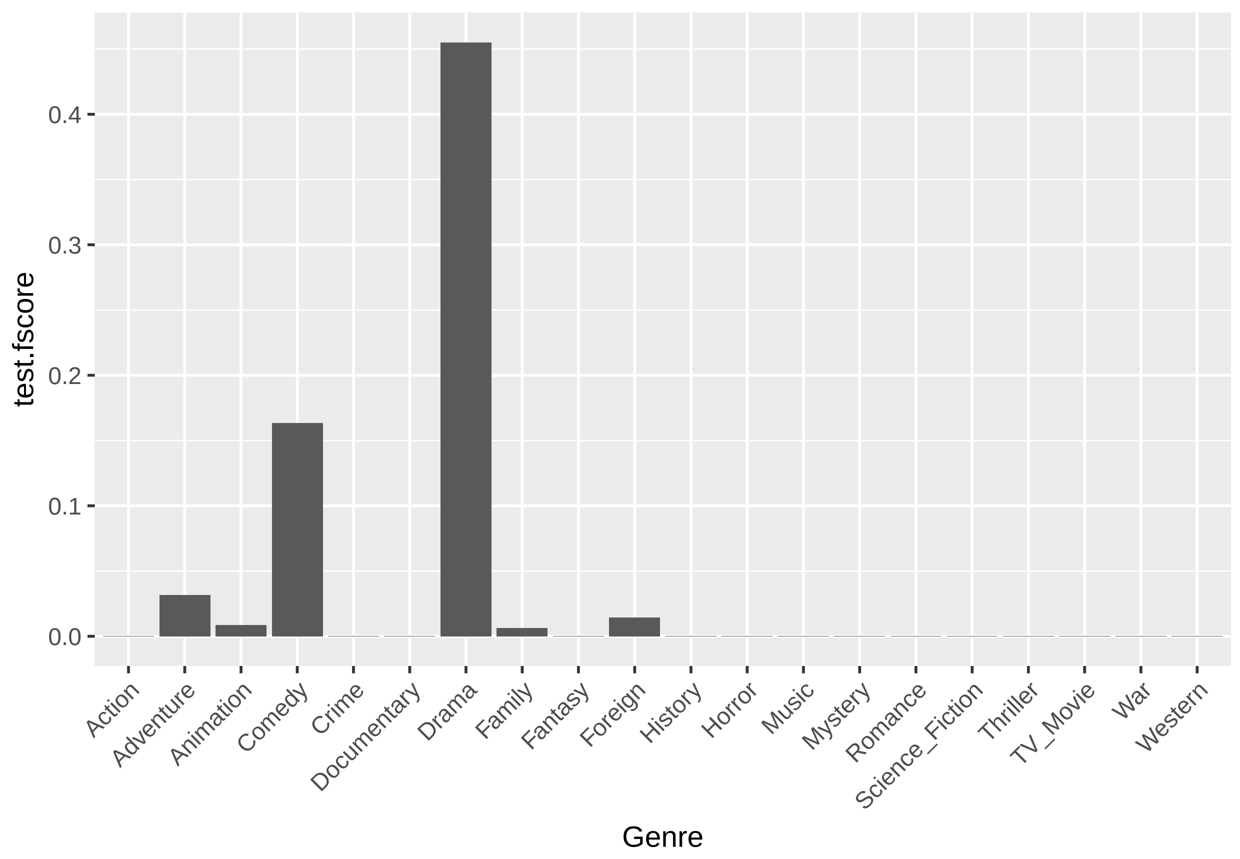
1. **Accuracy based model selection**

```
# find model with highest accuracy for each genre
max_accuracy = as.data.frame(as.data.table(mperf)[, .SD[which.max(train.accuracy)], by=Genre])
```

```
#plot accuracy for each selected model
ggplot(data = max_accuracy, aes(x = Genre, y = test.accuracy)) +
  geom_bar(stat="identity") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```
#plot fscore for each selected model
ggplot(data = max_accuracy, aes(x = Genre, y = test.fscore)) +
  geom_bar(stat="identity") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```

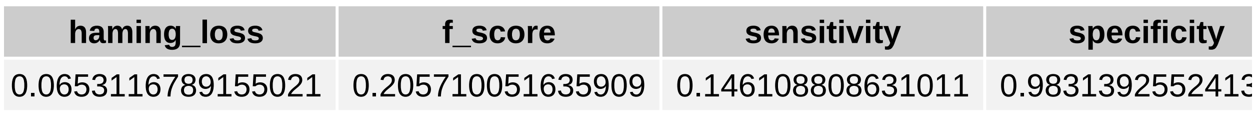


```
#Confusion Matrix
confusion_matrix_max_accuracy_test <-
  data.frame(prediction_1 = c(sum(max_accuracy$Test_TruePos),
                              sum(max_accuracy$Test_FalsePos)),
             prediction_0 = c(sum(max_accuracy$Test_FalseNeg),
                              sum(max_accuracy$Test_TrueNeg)),
             row.names = c('true_1', 'true_0'))
# plot
p<-tableGrob(confusion_matrix_max_accuracy_test)
grid.arrange(p)
```

	prediction_1	prediction_0
true_1	4442	25960
true_0	8343	486475

```
# ensemble model performance

#hamming loss
hamming_loss_test_accuracy_model <-
  (sum(max_accuracy$Test_FalsePos) +
   sum(max_accuracy$Test_FalseNeg))  /
  (sum(max_accuracy$Test_TruePos) +
   sum(max_accuracy$Test_FalseNeg) +
   sum(max_accuracy$Test_TrueNeg) +
   sum(max_accuracy$Test_FalsePos))

#f score
fscore_test_accuracy_model <-
  2 * sum(max_accuracy$Test_TruePos) /
  (2* sum(max_accuracy$Test_TruePos) +
      sum(max_accuracy$Test_FalsePos) +
      sum(max_accuracy$Test_FalseNeg))

# sensitivity
sensitivity_test_accuracy_model <-
  sum(max_accuracy$Test_TruePos) /
  (sum(max_accuracy$Test_TruePos) +
   sum(max_accuracy$Test_FalseNeg))

# specificity
specificity_test_accuracy_model <-
```

```
  sum(max_accuracy$Test_TrueNeg)/
  (sum(max_accuracy$Test_FalsePos) +
   sum(max_accuracy$Test_TrueNeg))
mperf_assembled_accuracy <- data.frame(
  haming_loss = hamming_loss_test_accuracy_model,
  f_score = fscore_test_accuracy_model,
  sensitivity = sensitivity_test_accuracy_model,
  specificity = specificity_test_accuracy_model)
p<-tableGrob(mperf_assembled_accuracy)
grid.arrange(p)
```

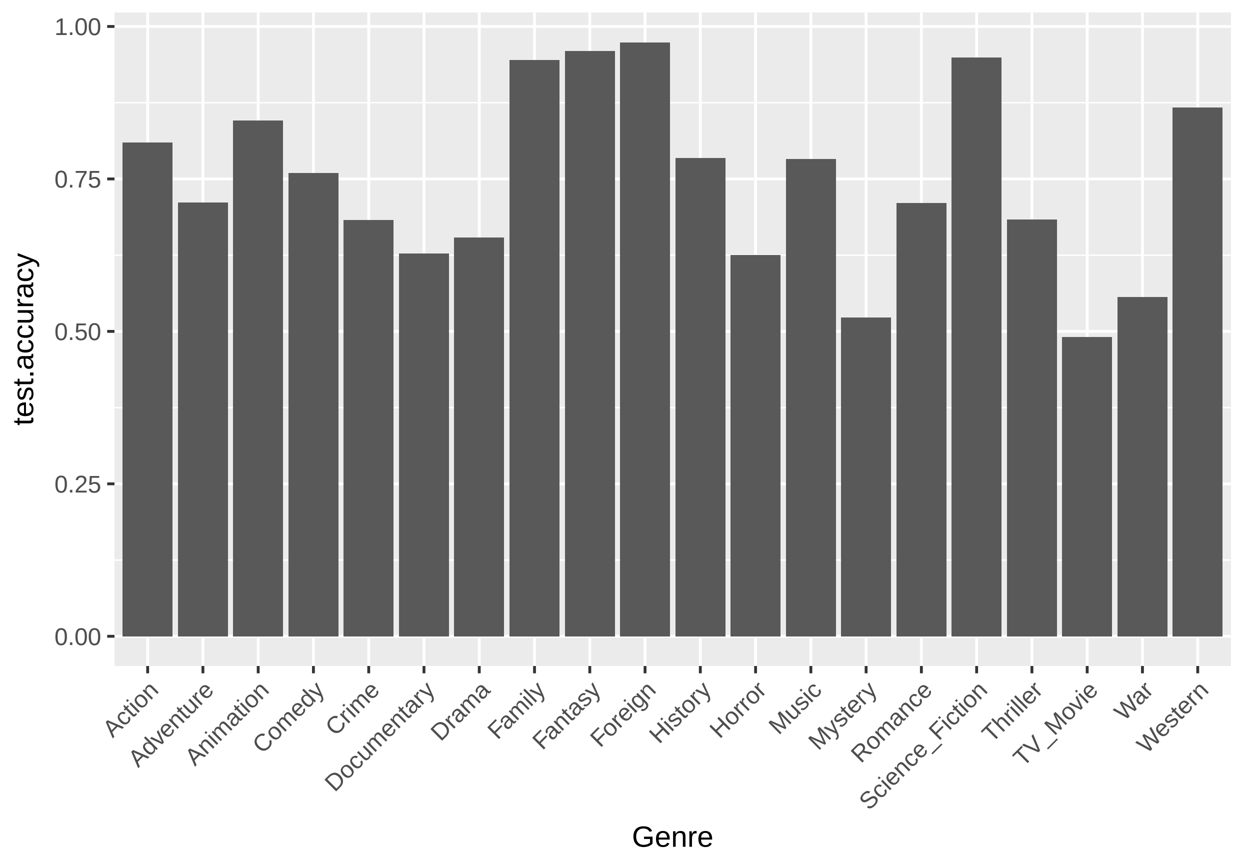


haming_loss	f_score	sensitivity	specificity
0.0653116789155021	0.205710051635909	0.146108808631011	0.9831392552413

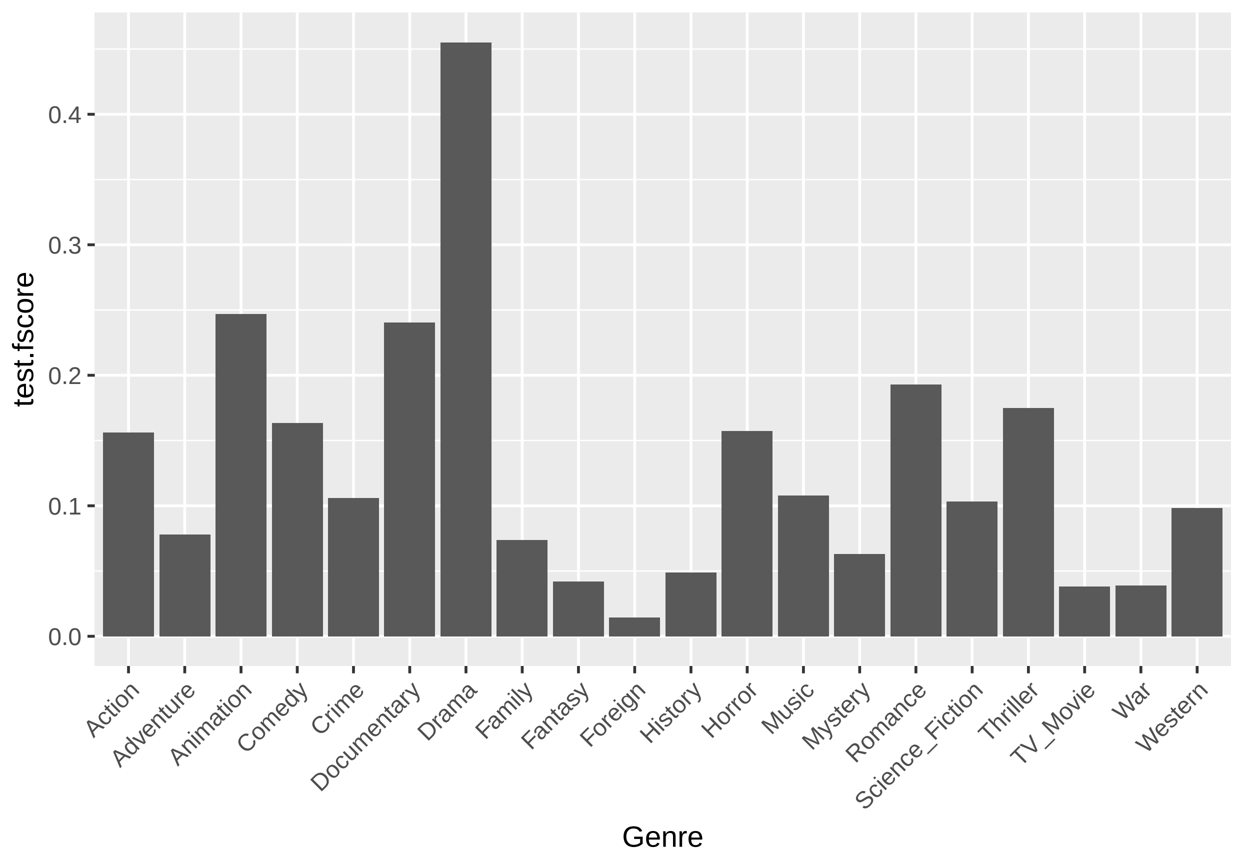
2.F1 score based model selection

```
# find model with highest f score for each genre
max_fscore = as.data.frame(as.data.table(mperf)[
  , .SD[which.max(train.fscore)], by=Genre])
```

```
# plot accuracy for each selected model
ggplot(data = max_fscore, aes(x = Genre, y = test.accuracy)) +
  geom_bar(stat="identity") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```r
# plot fscore for each selected model
ggplot(data = max_fscore, aes(x = Genre, y = test.fscore)) +
  geom_bar(stat="identity") +
  theme(axis.text.x = element_text(angle = 45, hjust = 1))
```



```r
#Confusion Matrix
confusion_matrix_max_fscore_test <- data.frame(
  prediction_1 = c(sum(max_fscore$Test_TruePos),
                   sum(max_fscore$Test_FalsePos)),
  prediction_0 = c(sum(max_fscore$Test_FalseNeg),
                   sum(max_fscore$Test_TrueNeg)),
  row.names = c('true_1', 'true_0'))

# plot
p<-tableGrob(confusion_matrix_max_fscore_test)
grid.arrange(p)
```

	prediction_1	prediction_0
true_1	12376	18026
true_0	114829	379989

```
# ensemble model performance
#hamming loss
hamming_loss_test_fscore_model <-
  (sum(max_fscore$Test_FalsePos) +
   sum(max_fscore$Test_FalseNeg))/
  (sum(max_fscore$Test_TruePos) +
   sum(max_fscore$Test_FalseNeg) +
   sum(max_fscore$Test_TrueNeg) +
   sum(max_fscore$Test_FalsePos))

#f score
fscore_test_fscore_model <-
  2 * sum(max_fscore$Test_TruePos) /
  (2* sum(max_fscore$Test_TruePos) +
      sum(max_fscore$Test_FalsePos) +
      sum(max_fscore$Test_FalseNeg))

# sensitivity
sensitivity_test_fscore_model <-
  sum(max_fscore$Test_TruePos) /
  (sum(max_fscore$Test_TruePos) +
   sum(max_fscore$Test_FalseNeg))

# specificity
specificity_test_fscore_model <-
  sum(max_fscore$Test_TrueNeg) /
```

```r
  (sum(max_fscore$Test_FalsePos) +
   sum(max_fscore$Test_TrueNeg))

mperf_assembled_fscore <- data.frame(
  hammming_loss = hamming_loss_test_fscore_model,
  f_score = fscore_test_fscore_model,
  sensitivity = sensitivity_test_fscore_model,
  specificity = specificity_test_fscore_model)
p<-tableGrob(mperf_assembled_fscore)
grid.arrange(p)
```

	hammming_loss	f_score	sensitivity	specificity
!	0.252951144282396	0.15704886204293	0.407078481678837	0.7679368980109

Compared with the ensemble model based on selecting models with the highest F1 score, the ensemble model based on selecting models with the highest accuracy results in a lower Hamming Loss, lower F1 score, lower sensitivity, and extremely higher specificity (approximately 1). In the accuracy-based model, almost all samples were classified as class 0, and very few as class 1.

Considering the imbalanced nature of data, excellent accuracy (higher than 90%) is only reflecting the underlying class distribution. So accuracy is misleading here and is not the metric to use.

The F1 score is the geometric mean of precision and sensitivity, and is a more reasonable metric to use for our dataset. Under the F1 score strategy, all the selected models are the "LogReg unbalanced w/H predictors and base_rate_cutoff". The individual model F1 scores are between 0.03 and 0.48. The accuracies are between 0.29 and 0.93.

There are several methods we can try to improve the performance on imbalanced data:

1. Resampling the dataset. We can try undersampling the majority class or oversampling the minority class.

2. Penalized model. For example, we can change the misclassification penalty per class in SVM (class weighted SVM)

In addition to the Hamming Loss, we can also consider the Macro-F1 loss, as described above:

```
macroF1_by_model <- aggregate(mperf$Test_F1,
                              by=list(mperf$Model_Name),
                              mean)
colnames(macroF1_by_model) <-
  c('ModelName', 'MacroF1')
levels(macroF1_by_model$ModelName) <-
  c(levels(macroF1_by_model$ModelName), 'Ensemble')
macroF1_for_ensemble <- mean(
  aggregate(mperf$Test_F1,
            by=list(mperf$Genre),
            max)$x)
macroF1_by_model <- rbind(macroF1_by_model, c('Ensemble',
                                              macroF1_for_ensemble))
macroF1_by_model  # print it...
```

```
##                                                ModelName
## 1                                      LogReg 3 predictors
## 2                          LogReg unbalanced w/3 predictors
## 3  LogReg unbalanced w/3 predictors and base_rate_cutoff
## 4                          LogReg unbalanced w/H predictors
## 5  LogReg unbalanced w/H predictors and base_rate_cutoff
## 6                 LogReg w/cast-score and overview_features
## 7                           LogReg w/cast-score predictors
## 8                                   LogReg w/H predictors
## 9                                           Most Frequent
## 10                                           RandomForest
## 11                                    RandomForest Fscore
## 12                                       RandomForest w/H
## 13                                             Stratified
## 14                                                    SVC
## 15                                             SVC Fscore
## 16                                                SVC w/H
## 17                                               Ensemble
##                MacroF1
## 1  0.00607106692679142
## 2  0.00607106692679142
## 3    0.133059522433159
## 4  0.00458378811448872
## 5    0.129130325068589
## 6  0.00587733810087036
## 7  0.00442750383583679
## 8  0.00458378811448872
## 9                    0
## 10  0.0668764830591864
## 11   0.149408870634128
## 12 0.00509942893931363
## 13  0.0566024541620821
## 14   0.130903552406827
## 15   0.137527054000514
## 16   0.138358725043845
## 17   0.166119163423742
```

Discussion of the differences between the models, their strengths, weaknesses, etc.

The basics of the models were discussed above.

The Logistic Regression models are quick to train but have only mediocre success. Not only are the classes for each genre highly imbalanced, but they are not terribly separable. Even when adding cast affinity scores and text features extracted from the overview, the F1 scores for the LR models were quite poor – except by comparison with the others. (See the F1 chart above, in which models "C" and "E" in general do relatively less poorly than the others.)

However, we believe that is because there was not enough time to properly tune the other models; the SVC model shows promise in some genres (such as for Animation, Comedy, Documentary, Romance, and Science Fiction, History, Mystery, and Western), in some cases nearly matching or even exceding the performance of the LR models. Given the time to try tuning it over a wider range of parameters, we believe SVC will do at least as well as Logistic Regression. (This is hardly a surprising suggestion, since LR is essentially a subset of what SVC can compute).

The Random Forest takes hours to tune, even with the train data throttled to 1%. As a result, we do not yet have confidence in our assessment of its performance. Drama is the only category in which its F1 score is not essentially 0, which indicates that our grid search was in the entirely wrong space of hyperparameters. The Random Forest class of model has the particular strength that it is quite effective in mixing predictors of widely different types – dollars (budget and revenue), long-term time (year of release) and short-term time (runtime in minutes), etc. For our purposes, though, its biggest drawback is the time needed to tune it.

One other family of models that we wanted to include but could not because of time is Generalized Linear Models. Experimenting with smoothed or polynomial predictors might have been productive.

Discussion of the performances you achieved, and how you might be able to improve them in the future

Possible future directions:

- While we have done a minimal amount of tuning our models, we did not have time to refine the search grid to truly optimize the hyperparameters.
- We did not have time this week to explore building GLMs using our predictors. It would be interesting to explore that, and especially to use ANOVA to compare GLMs that differ only in the inclusion/exclusion of a single predictor in turn, to assess the value of each predictor.
- In general. it would be useful to examine the relative strength of the predictors in our various models, to allow us to prune those that fail to add enough additional predictive power to justify the added complexity and computation time to our models.

Appendix

As always, our git repo can be found at https://github.com/amgreene/cs109b

A. Code for creating various genre affinity scores:

```r
library(dplyr)
library(readr)
library(tidyr)

imdb_sep <- read_tsv("../data/imdb_data_separated.txt", col_names = FALSE,
                     col_types = list(col_character(), col_character(), col_character(), col_character(
names(imdb_sep) <- c("IMDB_ID", "field_type", "field_id", "field_name")

imdb_tsv <- read_tsv("../data/imdb.tsv")
```

```r
fields <- c("directors", "production_companies", "cast", "writer",
            "producer", "composer", "miscellaneous_crew", "distributors",
            "editor", "cinematographer", "production_designer")

for (f in fields) {
  print(f)
  duplicate_directors <- imdb_sep %>%
  filter(field_type == f) %>%
  select(field_id, field_name) %>%
  unique() %>%
  group_by(field_id) %>%
  summarise(count = n()) %>%
  ungroup() %>%
  filter(count == 2) %>%
  .$field_id

  imdb_directors_movie_genre <- imdb_tsv %>%
  select(IMDB_ID, `Sci-Fi`:Biography) %>%
  right_join(imdb_sep %>% filter(field_type == f, !field_id %in% duplicate_directors))

  imdb_directors_genre_affinity <- imdb_directors_movie_genre %>%
  select(`Sci-Fi`:Biography, field_id, field_name) %>%
  gather(genre, genre_score, `Sci-Fi`:Biography) %>%
  group_by(field_name, field_id, genre) %>%
  summarise(genre_affinity = sum(genre_score)) %>%
  ungroup() %>%
  filter(genre_affinity > 10) %>%
  spread(genre, genre_affinity)

  movie_genre_affinity_score <- imdb_sep %>%
  filter(field_type == f) %>%
  inner_join(imdb_directors_genre_affinity) %>%
  select(IMDB_ID, Action:War) %>%
  group_by(IMDB_ID) %>%
  summarise_all(.funs = funs("sum"))

  movie_genre_affinity_score[is.na(movie_genre_affinity_score)] <- 0

  movie_genre_affinity_score %>%
  write_tsv(paste0("../data/imdb-movie-", f, "-scores.tsv"))
  print(f)
}
```

B. Code for feature extraction from movie overview.

```python
# This script creates text feaures from overview of movies in tmdb

import pandas
import nltk
# nltk.download() # to dowload the corpus
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import re
import string
import pycountry
from json import loads

dataset = pandas.read_table('../data/tmdb.tsv',
                               na_values = 'None')

ids_dedup = set()
train_movies = []
for year in xrange(2017, 1914, -1):
    for movie_str in open('../data/tmdb/tmdb-details-' + str(year) + '.txt', 'r'):
        movie = loads(movie_str)

        # Do the dedup
        if movie['id'] in ids_dedup:
            continue
        ids_dedup.add(movie['id'])

        movie['tmdb_id'] = movie['id']
        if movie['tmdb_id'] in dataset['tmdb_id']:
            train_movie = {
                'tmdb_id': movie['tmdb_id'],
                'overview': unicode(movie['overview'])
            }
            train_movies.append(train_movie)

print "Tokenizing movie oveview"
# tokenize movie overview
tokenized_docs = [word_tokenize(doc['overview'].lower()) for doc in train_movies]

print "Removing punctuations"
# remove punctuations
regex = re.compile('[%s]' % re.escape(string.punctuation))
tokenized_docs_no_punctuation = []
for overview in tokenized_docs:
    new_overview = []
    for token in overview:
        new_token = regex.sub(u'', token)
        if not new_token == u'':
            new_overview.append(new_token)
    tokenized_docs_no_punctuation.append(new_overview)

print "Removing stopwords"
# remove stopwords from overviews in these languages:
stopwords_all_language = []
nltk_lang = ['danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian', 'italian',
        'kazakh', 'norwegian', 'portuguese', 'russian', 'russian', 'spanish', 'swedish']
for lang in nltk_lang:
    stopwords_all_language.extend(stopwords.words(lang))

tokenized_docs_no_stopwords = []
for doc in tokenized_docs_no_punctuation:
    new_term_vector = []
    for word in doc:
        if not word in stopwords_all_language:
            new_term_vector.append(word)
    tokenized_docs_no_stopwords.append(new_term_vector)

print "Populating most common keywords"
# populate the top 50 keywords by frequency
from collections import Counter
tf = Counter()
for doc in tokenized_docs_no_stopwords:
    for word in doc:
        tf[word] +=1
most_common_words = [word for word, word_count in tf.most_common(50)]
print most_common_words

print "Writing to file."
# write them to file
with open('../data/overview_features.tsv', 'w') as o:
    # Write TSV header row
    o.write('tmdb_id' + '\t')
    o.write('\t'.join(most_common_words))
    o.write('\n')

    for idx, doc in enumerate(tokenized_docs):
        o.write(str(train_movies[idx]['tmdb_id']) + '\t')
        o.write('\t'.join(['1' if words in doc else '0' for words in most_common_words]))
        o.write('\n')

# create the same features for test set:
test_dataset = pandas.read_table('../data/tmdb-test.tsv',
                                 na_values = 'None')
ids_dedup = set()
test_movies = []
for year in xrange(2017, 1914, -1):
    for movie_str in open('../data/tmdb/tmdb-details-' + str(year) + '.txt', 'r'):
        movie = loads(movie_str)

        # Do the dedup
        if movie['id'] in ids_dedup:
            continue
        ids_dedup.add(movie['id'])

        movie['tmdb_id'] = movie['id']
        if movie['tmdb_id'] in test_dataset['tmdb_id']:
            test_movie = {
                'tmdb_id': movie['tmdb_id'],
                'overview': unicode(movie['overview'])
            }
            test_movies.append(test_movie)

# tokenise overview in test set
test_tokenized_docs = [word_tokenize(doc['overview'].lower()) for doc in test_movies]

# write them to file
```

```
with open('../data/overview_features-test.tsv', 'w') as o:
    # Write TSV header row
    o.write('tmdb_id' + '\t')
    o.write('\t'.join(most_common_words))
    o.write('\n')

    for idx, doc in enumerate(test_tokenized_docs):
        o.write(str(test_movies[idx]['tmdb_id']) + '\t')
        o.write('\t'.join(['1' if words in doc else '0' for words in most_common_words]))
        o.write('\n')

print "Feature creation complete"
```