CS 361: Final Report: A Content-Based Filtering Approach for Predicting Games of Interest for a User on Steam

Arvind Vijayakumar, Harris Lynch, and Thomas Yu

May 17th, 2016

Abstract

In this paper, we describe a content-based filtering approach to predicting games that a given user of Steam (a digital distribution platform) may want to play/purchase given the games currently in their account. We will look at a k-Nearest Neighbors approach of the problem, and a ranking formula for the games in an account. We will also look at a method of generating a "cold start", or zero information recommendation of games for users without any games in their account (ie. new users). Finally, we will analyze the performance of the recommender, and look at possible improvements to the algorithm.

1 Introduction

Steam is digital distribution platform primarily used for distributing local and multiplayer video games. As such there are large number of games on the market, and so for a user, finding interesting games that he or she might like to play can become difficult if the most popular, well-known games in their categories (eg. the *Counter-Strike* franchise, *Dota 2*, the *Portal* games, etc.) have already been considered. Steam does provide a tool to solve this problem, their "Discovery Queue", however, the recommended games provided by the discovery queue are often either dissimilar to a user's favorite games, or more crucially, the user has no interest in playing them. We attempt to solve this problem by creating our own recommender using a naive approach: finding similar games to a user's favorite games.

2 Algorithm

We employed a k-Nearest Neighbors algorithm or Nearest Neighbor algorithm for each of the user's ten favorite games to generate ten recommendations. If the user had n games, where $n < 10$, then only n recommendations were made. Using a nearest neighbor algorithm for only the favorite games guarantees consistent behavior whether or not n is very large (eg. 1000+) or very small (eg. 0-10).

Our database of games was constructed by scraping metadata from the official Steam api of the top 800 most popular games along with other associated metadata from external services like *Steam Spy*, and *SteamDB*. Choosing this subset of the total data optimized the time spent on each nearest neighbor search at the minimal risk that perhaps there is a very poorly known game that perfectly matches the user's favorite games. We performed basic imputations on the data by replacing NA fields for the "Developer", "Publisher", and "Description" fields with empty strings. We used TF-IDF weighting on the "Description" field of each game as well as a one-hot encoding from tags and genres to extract features from the data. We also experimented with column weighting, specifically with "Tag" columns (ie. Action, or Sports), eventually settling on weighting all "Tag" columns by 2x. This significantly helped distinguish sandbox games from other games, because sandbox games were more difficult to tell apart simply by the TF-IDF weighting of the descriptions. Our nearest neighbors algorithm used the euclidean distance between these final feature vectors to find similar games.

2.1 Ranking

Our ranking algorithm sought to determine a user's favorite games based solely off of their Steam user profile data. A user's profile data has the number of games they own, and a list of all their games, along with their recent playtime (playtime in the last two weeks, t_w) and their total playtime (t_f) for each game. We applied basic imputations by replacing missing playtime values with zero. Our logic behind ranking favorite games was that a user's current favorite, or most interested game should be one that he has played a lot recently, but has also put a fair amount of time in total. Correspondingly we developed two logarithmic ranking formulas that correlated the recent and total playtimes: a "lowend bias" (1) that favors games that you have played more recently, and a "highend bias" (2) that favors games that you have played more in total.

$$w_{lb} = (1 + 2\log_2(t_f + 1)) * (1 + 0.5\log_8(t_w + 1)) \tag{1}$$

$$w_{hb} = (1 + 6\log_2(t_f + 1)) * (1 + 0.16\log_6 4(t_w + 1)) \quad (2)$$

We empirically determined that using "lowend bias" as the weighting scheme, and then sorting your favorite games based off of the calculated weights produced the best or most convincing results.

3 Cold Start

3.1 Defining the Problem

"Cold start" is a term that is generally used to describe a problem in data modeling where a given data points (eg. a steam user) has no information, and can therefore not be acted upon using conventional methods. Since our algorithm is purely game based, a cold start situation arises whenever a user with no games is entered into the algorithm. The principle behind our approach to this problem is very simple: if a user has just downloaded or began using Steam, they will most likely want to play a game that many people like. How do you determine the favorite game of a community like Steam? Let us consider a few different possible ways: The game with the highest owner count could be chosen, but that would be biased towards older games like *Counter-Strike* or free games like *Team Fortress 2* or *Dota 2*. Note, all three of these games are in the top 10 for most owners. A game could be chosen by looking at mean/median playtimes, but that could be as dependent on the type of game. For example, *Knight Online* has the highest mean playtime currently on Steam (according to Steam Spy), yet, it only has 78,297 players and a relatively poor Metascore of 43% in sharp contrast to a game like Dota 2, which has in excess of 9 million players, and a Metascore of 90%. In regards to median playtime, *Grand Theft Auto V* is one of the most popular games on the market, but it has a median playtime of 2:33, far off from the highest median playtimes. As such, it becomes apparent that a suggestion of "the game to buy when you first buy Steam" must be determined by a multitude of factors. To make the problem more interesting, instead of periodically calling APIs and doing some sort of static analysis on the games, we decided to solve this by using a persistent model that keeps track of all the games it sees when it explores user's profiles, and updates itself in such a way that the more popular games will naturally bubble up to the top of the heap(so to speak).

3.2 Model

Our model is based off of the principle that if our algorithm sees more instances of a certain game in people's favorite games, then that game should be the best suggestion. It also takes into consideration the number of owners associated with each game, because if not many people own the game, then it cannot truly be considered "THE" game to get. From a programming standpoint, the "model" is an object in Python that is deserialized when necessary, updated, and serialized back to disk. The model stores a score for every visited game in a dictionary, and updates the score whenever it looks at a new user's account. It updates the scores by using both ranking weights (higher for a given user's more favored games) ranging from 1 to 1.5 and a logarithmic formula that incorporates both the number of owners of the game, and the number of games the user has (ie. the more games a user has the higher his credibility/the more significant his choice of favorite game is) (3). The number of owners is denoted as n_o, and the number of games is denoted as n_g. To remain efficient, the object maintains references to the current top-ranked game and its score. When it runs an update, it only compares the updated score to the highest score. Ties are broken by number of owners (ie. a game with more owners than another will be ranked higher if their scores are tied).

$$v_{update} = (\log_{64} n_o) * (\log_{10} n_g) \tag{3}$$

The model is initialized with every possible game as zero, except for a realistic starting guess of a popular game, which has a score of 1 (an inconsequential score in the long run, but enough for it to start out as the best game). In our case, the model was initialized with *Counter-Strike: Global Offensive.*

4 Testing Platform

We tested our algorithm by building a Flask application that takes in a user profile URL as input from an HTML form. The Flask app passes in the user steam64 UID as input to our algorithm's python script. The python script returns a JSON that is parsed and then displayed on the webpage.

5 Current Performance and Future Improvements

Our algorithm was tested by inputing a wide variety of both known and random users, and querying them about the accuracy of the results, or if unresponsive, gaging the results on our own. In the common case (ie. not cold start), the results were generally believable, understandable, and realistic. Unfortunately, one potential problem that we consistently ran into was the algorithm predicting other, often earlier, games in a series. In this regard, some sort of optional series detection would be very useful, because for example, if a user already has *The Elder Scrolls V: Skyrim* and *The Elder Scrolls IV: Oblivion*, there is a pretty decent chance that they are already aware of *The Elder Scrolls III: Morrowind* even if they do not currently own it. We were unable to find a dataset, API, or information source online that mapped different games in a series to each other, so we attempted to build our own series detection disjoint sets structure. Our approach was to label a small subset of the data, train a classifier on this subset, and then run it against the entire dataset, and use it to label the rest. Unfortunately, we were unable to finish due to time restrictions. In the cold start case, the results were generally pretty realistic. After 5-8 iterations on most sets of decently experienced users, the model had "bubbled up" fairly popular, and well-critically received games such as *Counter-Strike: Global Offensive*, *Team Fortress 2*, and *Portal 2*.

One major improvement to the entire algorithm would be to incorporate not only a content-based filtering system (ie. the game data of each user), but also a community-based filtering system. Steam has a well established system for connecting with or "friending" other users. Steam can very often be a social experience where users will buy games based off of what friends in real life have bought, and incorporating that aspect into the algorithm would certainly improve the performance. Furthermore it would lower the chance of encountering a cold start case, because in this scenario, a cold start user would have to both have no games AND have no other users connected to them on Steam.

One minor improvement could be made with the way results are chosen using the k-Nearest Neighbors Algorithm. Given the relative scores of the user's top 10 favorite games, multiple games could be chosen from one set of nearest neighbors, rather than one (the closest) from each game's set. The best manner in which these games could be chosen or appropriated between a chosen set, and a rejected set is still unclear to us.

A potential improvement in performance could also be found in adapting a more nuanced approach and using some sort of clustering methods such as k-Means, and generating recommendations based off of a certain cluster or a certain set of clusters.

1 Appendix: Algorithm Source Code

1.1 recommend.py

```
import numpy as np
import pandas as pd
import requests

import scipy.sparse as spp
from scipy.spatial import distance
from sklearn.neighbors import NearestNeighbors
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer

import time
import math

from xml.etree import cElementTree as ET

import pickle
# import cPickle as pickle
import os.path
from model import ModelPersist

class CONST(object):
    API_KEY = "<API_KEY>"
    DB_SOURCE = "./static/db.csv"
    MODEL_SOURCE = "./static/model.p"
    def __setattr__(self, *_):
        pass

CONST = CONST()

def get_steamID64(url):
    url_elements = url.split("/")
    print url_elements
    if 'home' in url_elements:
        url_elements.remove('home')
    url = '/'.join(url_elements)
    if url[-1] != '/':
        url += '/'
    url_string = url + '?xml=1'
    print url_string
    page = requests.get(url_string)
    page.raise_for_status()
    root = ET.fromstring(page.content)
    element = root.getchildren()[0]
    if element.tag == 'steamID64':
        return int(element.text)
    for ele in root.getchildren():
        if ele.tag == 'steamID64':
            return int(ele.text)
    return -1

def check_cold_start(url):
    r = requests.get(url)
    if r.json()['response']['game_count'] == 0:
        return True
    return False

def get_user_json(url):
    r = requests.get(url)
    # if r.json()['response']['game_count'] == 0:
    #     print 'No games on the account'
    #     #raise Exception('No games')
    return r.json()
```

```python
def get_user_data_from_json(json_file):
    temp = pd.DataFrame(json_file['response']['games'])
    if 'playtime_2weeks' not in temp.columns:
        temp['playtime_2weeks'] = 0
    temp.columns = ['AppId','playtime_2weeks','playtime_forever']
    return temp

def get_user_data(url):
    temp = pd.DataFrame(get_user_json(url)['response']['games'])
    if 'playtime_2weeks' not in temp.columns:
        temp['playtime_2weeks'] = 0
    temp.columns = ['AppId','playtime_2weeks','playtime_forever']
    return temp

def weight(t_f,t_w):
    return weight_lowend_bias(t_f,t_w)

def weight_lowend_bias(t_f,t_w):
    return (1.0+2.0*math.log(t_f+1.0,2.0))*(1.0+0.5*math.log(t_w+1.0,8.0))

def weight_highend_bias(t_f,t_w):
    return (1.0+6.0*math.log(t_f+1.0,2.0))*(1.0+0.16*math.log(t_w+1.0,64.0))

def generate_weights_dict(df):
    weights = {}
    for i,row in df.iterrows():
        #print type(row)
        t_w = row['playtime_2weeks']/60.0
        t_f = row['playtime_forever']/60.0
        weights[row['appid']] = weight(t_f,t_w)
    return weights

def generate_weights(df,remove_columns=False):
    df['Weight'] = 0.0
    for i,row in df.iterrows():
        #print type(row)
        t_w = row['playtime_2weeks']/60.0
        t_f = row['playtime_forever']/60.0
        df.set_value(i,'Weight',weight(t_f,t_w))
    if remove_columns:
        df.drop(['playtime_2weeks'],axis=1,inplace=True)
        df.drop(['playtime_forever'],axis=1,inplace=True)
    return weight

def read_database(file=CONST.DB_SOURCE):
    return pd.read_csv(file)

def misval(df,columns=[],categorical=False):
    if categorical:
        for col in columns:
            df[col].fillna("",inplace=True)
    else:
        for col in columns:
            df[col].fillna(0,inplace=True)

def one_hot_encoding(df,columns=[],sep="*"):
    for col in columns:
        df = df.join(pd.Series(df[col]).str.get_dummies(sep=sep))
        df.drop([col],axis=1,inplace=True)
    return df

def cold_start():
    if os.path.isfile(CONST.MODEL_SOURCE) :
        model = pickle.load(open(CONST.MODEL_SOURCE, 'rb'))
        return model.get_tuple()
    else:
        model = ModelPersist()
        ret = model.get_tuple()
        pickle.dump(model, open(CONST.MODEL_SOURCE, 'wb'))
        return ret

# @param: uid as numeric
# @return: list of tuples: ( appid(owned_game), name(owned_game),
  appid(predicted_game), name(predicted_game) )
def recommend(uid):
    # -->Setting up & Preprocessing Data<--
    # load sqlserver data
    db = read_database()
    # clean up / preprocess server data
      (Note: perhaps implement data cleaning upon storage into server)
    misval(db,columns=['Developer','Publisher','Description'],categorical=True)
    db = one_hot_encoding(db,columns=['Genre','Tag'],sep="*")
    db = pd.get_dummies(db,columns=['Developer','Publisher'])
    # load user data
    access_string = 'http://api.steampowered.com/IPlayerService/GetOwnedGames/v0001/?key='
                    +CONST.API_KEY+'&steamid='+str(uid)+'&format=json&include_played_free_games=true'
    if check_cold_start(access_string):
        return [cold_start()]
    json_file = get_user_json(access_string)
    game_count = json_file['response']['game_count']
    user_data = get_user_data_from_json(json_file)
    # generate weights column + remove playtime columns
    misval(user_data,columns=['playtime_2weeks','playtime_forever'],categorical=False)
    generate_weights(user_data,remove_columns=True)
    # sort list by weights + remove weight column
    user_data.sort_values('Weight',axis=0,ascending=False, inplace=True,
      kind='quicksort', na_position='last')
    user_data.drop(['Weight'],axis=1,inplace=True)
    # merge user_data with db
    user_data = pd.merge(user_data, db, how='inner', on=['AppId'])
    # create dictionary of owned games
    owned_games = set(user_data['AppId'].to_dict().values())
    if len(user_data.index) > 10:
        user_data = user_data.iloc[0:10]

    # Update persistent 'cold start' model
    if os.path.isfile(CONST.MODEL_SOURCE):
        model = pickle.load(open(CONST.MODEL_SOURCE, 'rb'))
    else:
        model = ModelPersist()

    model.update(user_data, game_count, uid)
    pickle.dump(model, open(CONST.MODEL_SOURCE, 'wb'))

    # Tf-idf word processing

#   Column Weighting
    for col in db.columns:
        if 'Tag' in col:
            db[col] *= 2

#     db['TagSandbox'] *= 2.5

    for col in user_data.columns:
        if 'Tag' in col:
            user_data[col] *= 2

#     user_data['TagSandbox'] *= 2.5
```

```
    desc_vector = TfidfVectorizer(analyzer='word', ngram_range=(1,1),
      stop_words = 'english')
    db_word_matrix = desc_vector.fit_transform(db['Description'])
    user_word_matrix = desc_vector.transform(user_data['Description'])

    title_vector = TfidfVectorizer(analyzer='word', ngram_range={1,2},
      stop_words = 'english')
    db_title_matrix = title_vector.fit_transform(db['Title'])
    user_title_matrix = title_vector.transform(user_data['Title'])

    # Temp Drop
    db.drop(['Owners'],axis=1,inplace=True)
    user_data.drop(['Owners'],axis=1,inplace=True)

    # Convert Data to sparse matrices and hstack with Tfidf features
    db_num = db._get_numeric_data().as_matrix()
    user_num = user_data._get_numeric_data().as_matrix()

    x_data = db_num[ : , 1: ]
    y_data = user_num[ : , 1: ]

    x_data_sparse = spp.csr_matrix(x_data)
    y_data_sparse = spp.csr_matrix(y_data)

    x_data_final = spp.hstack([x_data_sparse,db_word_matrix,db_title_matrix])
    y_data_final = spp.hstack([y_data_sparse,user_word_matrix,user_title_matrix])

    # -->Analyzing Data<--

    nbrs = NearestNeighbors(n_neighbors=40, algorithm='brute',metric='euclidean').fit(x_data_final)
    distances, indices = nbrs.kneighbors(y_data_final)
    chosen = []
    chosen_list = []
    for i in xrange(indices.shape[0]):
        n_iter = 0
        start_member = db.iloc[indices[i,0]]
        chosen_member = db.iloc[indices[i,1]]
        while chosen_member['AppId'] in owned_games or chosen_member['AppId'] in chosen_list:
            n_iter += 1
            chosen_member = db.iloc[indices[i,n_iter]]

        chosen_list.append(chosen_member['AppId'])
        chosen.append((start_member['AppId'],start_member['Title'],
          chosen_member['AppId'],chosen_member['Title']))
    return chosen
```

1.2 model.py

```
from collections import defaultdict
import numpy as np
import pandas as pd
from math import log

class ModelPersist:
  score = defaultdict(lambda: 0)
  ranking_weights = np.arange(1,1.5,0.05)
  visited = set()
  def __init__(self):
    #self.score = defaultdict(lambda: 0)
    self.score[730] = 1.0
    self.bestgame = 730
    self.num_owners = 0
    self.highest_score = 1
    self.names = {730 : 'Counter-Strike: Global Offensive'}

  def update(self, df, game_count, uid):
```

```python
    if( uid in self.visited):
      return
    self.visited.add(uid)
    for index, row in df.iterrows():
      incr = log(int(row['Owners']),64.0)*log(game_count,10.0)*1.0
      if index < 10:
        incr *= self.ranking_weights[9-index]
      self.score[row['AppId']] += incr
      self.names[row['AppId']] = row['Title']
      # print "current score: %f | current best: %f" % (self.score[row['AppId']], self.highest_score)
      # print "current game: %s | best game: %s" % (row['Title'],self.names[self.bestgame])
      # print "current num_owners: %d | best num_owners: %d" % (int(row['Owners']), self.num_owners)
      if float(self.score[row['AppId']]) > float(self.highest_score):
        self.highest_score = self.score[row['AppId']]
        self.bestgame = row['AppId']
        self.num_owners = row['Owners']
      if float(self.score[row['AppId']]) == float(self.highest_score):
        if int(row['Owners']) > self.num_owners:
          self.highest_score = self.score[row['AppId']]
          self.bestgame = row['AppId']
          self.num_owners = row['Owners']

  def get_highest_score(self):
    return self.highest_score

  def print_all_scores(self):
    for key, value in self.score.iteritems():
      print "key: %d | name: %s | value: %d" % (key, self.names[key], value)

  def get_best_game(self):
    return self.best_game

  def get_tuple(self):
    return (404, "err:nogame", self.bestgame,self.names[self.bestgame])
```