The Prediction of Movie Ratings using Random Forests

Dor Baruch, Peter Chang, Zachry Bai, Annie Hwang

Demo Video:

Overview and Instructions:

Our project predicts the ratings of movies considering attributes of runtime, genre, MPAA rating, budget, and gross box office income. We used two machine learning algorithms: neural networks and random forests.

Planning:

We planned on creating our project according to this draft specification and final specification. We planned to store the data extracted from IMDb into a matrix, with each row representing a dataset, and each column representing the attributes mentioned above. Using the regression model, we considered using decision trees that would at each node, check the purity of each subset by evaluating a boolean expression (i.e. rating_for_movie_of_director > 8.0) for all of the attributes in the order of dominance according to the ID3 algorithm.

However, we realized that with just decision trees, there would be problems of overfitting; so, we then planned to use random forests to accommodate this common problem with decision trees. With a random forest, we planned to divide the dataset to sub-datasets, and attributes to sub-attributes, and randomly selecting these sub-attributes to form attribute specific decision trees. Eventually, we planned to average the results at the leaves to predict the new ratings of movies. Unfortunately, we were unable to use some of the attributes (actors, producers, writers, plot summary, etc) that we planned to use for reasons that are detailed in the "Design and Implementation" section below.

Design and implementation:

The whole project was coded in Python. Our group used the imdb python package (http://imdbpy.sourceforge.net/) to query and store data (budget, actors, etc) into a CSV file. We were able to extract the data that we needed from the IMDb database and attempted to put it into a matrix for our random forest. Our group tried to store this data into a matrix, with each row representing a dataset, and each column representing the attributes that would affect what we were attempting to predict.

But some questions we had while implementing the matrix were:

1. How do we solve the problem of all movies having a different number of values for such attributes as writers, actors, and producers?
2. Should we separate different people of the same subfield (i.e. actor or writer) in different columns or put all of them in a single column with their quantitative value evaluated?

Along with these problems, we also realized that the unset number of actors, producers, directors, and other attributes that had multiple values would create too big of a dataset, which would then raise issues of being computationally infeasible not only to extract, but also to implement our decision trees.

Hence, we changed our project so that the program would only use attributes with specific quantitative or distinct values (such as runtime, genre, MPAA rating, budget, and rating) to predict the rating of movies. We implemented this by two different machine learning algorithms:

1. Neural Network - Our algorithm works in the following way:
    a. the __init__ function takes a list of A_i matrices and creates a NeuralNetwork with i layers, each made of A_i number of nodes. In this function the weight matrix ("weights") is initialized with random values according to the shape of the network.
    b. The "run" function then runs the input matrix through the network. The input matrix consists of 1500 rows of movies where the columns represent the following information about each movie: budget, genre, MPAA rating, runtime and gross box-office income. This function outputs a matrix with the resulted gross income. Using the .dot function from the numpy library, we perform a matrix multiplication between the weights matrix and each input row (where each row represents a different movie). We add the resulted matrix (which is a column vector) to the layerIn variable. We also run the our activation function (sigmoid) on that matrix, and add it to the layerOutput variable. This variable holds the outputs for each layer in the network.
    c. The role of the "backPropagate" function is to calculate the error and update the weights in the network to minimize it. The function takes as input the input matrix with all the movies, the matrix of expected ratings from the imdb database and a constant called learning_rate that reflects how fast we want the network to "learn". The function first calculates the deltas of the last layer (the output layer) by subtracting the desired output with the actual output from running the network. It also saves the the sum of the square of those deltas which represents our current error. Next the last layer's deltas are multiplied by the derivative of the sigmoid function evaluated at that layer's input and appended to the deltas list. For any other layer besides the last layer the deltas are calculated by performing a matrix multiplication between the last layer's deltas and the current layer's weights. Once again those deltas are multiplied by the derivative of the sigmoid function evaluated at that layer's input and appended to the deltas list. The next stage is using the deltas we calculated to adjust the weights. We created a new list variable called "wdelta" and appended to it the matrix multiplication of the daltas of each layer times the input to that layer.
    d. The next function is called update_wights. This function takes as input the variable "wdelta" that was created in "backPropogate" and the variable learning_rate. Each weights layer is then updated by subtracting each layer of the "wdelta" variable times the learning_rate from the current weights layer.
    e. The last function is called "train". This function takes as input the input list of movies, the target values from the imdb database, the learning rate and the number of times to run the network. The role of this function is to run the network

with the requested number of times, and report the error every 5000 runs through the network.

2. Random Forests- This was implemented much like how it is described above (in the “Planning” section) for our initial plan. In detail:
    a. Each non-leaf node in the decision trees consisted of a boolean test, which led the data sample to either the left or right child node depending on the result of the test, which were created in a manner to guarantee that the dataset would be the most well “separated”. To do so, we found the set with the smallest variance.
    b. We created the leaves for the binary trees when the maximum depth, which we have specified, was reached for the decision tree or when the number of patches left was minimal. Each leaf represents the average of all the attributes (budget, rating, etc) and offset vectors that led to this final leaf.
    c. Finally, to predict the box office success for a new movie, the program iterates the movie through the user-specified number of trees and averages the result to get the prediction.
    d. The resulting accuracy was about 88% accuracy, which was satisfactory.

Reflection

Due to scikit-learn library being open-source, it allowed us to understand the underlying principles in implementing decision trees and random forests. However, we realized that even with this open resource, we didn’t really know where or how to start. We had to spend an inordinate amount of time to understand the forest.py and tree.py in the scikit opensource libraries.

Our initial decision to implement trees with attributes that did not have quantitative values did not work out too well, because we didn’t have a clear grasp of how to assess the quantitative value of these attributes while still keeping the core concept of machine learning.

Our decision to change the purpose of the project to predict the box office success simplified our project a lot more because we did not have to assess the quantitative value of the string attributes, such as the actors, in terms of how it would affect the rating of the movie. Yet, we were still able to learn a significant amount of decision tree, the regression model, and random forests while implementing our program.

If we had more time, we would’ve liked to implement the naive bayse algorithm to compare not only the simplicity in coding a naive based algorithm, but also the accuracy and precision of our predictions. If we were to redo this project from scratch, we would still program in python and use neural networks, but we would reconsider using other implementations of our decision trees.

Advice for future students

We advise future students to seek out for help from their TF whenever there is a problem they encounter.