EE627-WS Data Acquisition and Processing I Final Project

Star Wars

Justin John, Ismet Okan Celik, Anthony Donatelli

Prof. Wang

05/05/21

Table of Contents

1. Summary of the methods you have tried (motivations, formula and discussions) 3
Simple Averages: 3
Weighted Averages: 3
PCA: 3
Matrix Factorization: 4
Logistic Regression: 5
Decision Tree: 5
Random Forest: 6
Gradient Boosted Tree: 6
Ensemble Testing: 7
2. The performance you observed and a brief explanation. 9
Simple Averages Score: 9
Weighted Averages Score: 9
PCA Score: 9
Matrix Factorization Score: 9
Logical Regression Score: 9
Random Forest Score: 10
Gradient Boosted Tree Score: 10
Decision Tree Score: 10
Ensemble Testing Score: 10
3. The overall results after ensembling and your comments. 10
4. Attach the script codes in separate files: 11

1. Summary of the methods you have tried (motivations, formula and discussions)

Simple Averages:

Our team started this project as simply as we could with a simple average of the predictor values. We began with using the AlbumId score and the ArtistId score, adding them together, then dividing by two. The score we received was not very good but it was a good start to the project and how well we could work together and share ideas.

Weighted Averages:

We started the project by trying to extract TrackID, Album, Artist, and Genres Scores. In the beginning, we just used TrackID and AlbumID rates to submit. First, we took the average of the rates we extracted from the data and made our first submission. After that, we thought we could improve our score using a different method, such as giving different importance to the features (AlbumID, ArtistID, e.t.c.)

Our first approach in terms of weighted average was: (w1*AlbumID+w2*ArtistID)/2=Final Rate

w1: Weight-1

w2: Weight-2

w1+w2=1

After applying this approach, we changed the weights in different combinations, took the average, and made different submissions on the Kaggle. Later on we try to extract other features (Genre1,Genre2,Genre3,e.t.c) to our calculations. When we include more features in our calculations, we add more weight values to the calculation.

PCA:

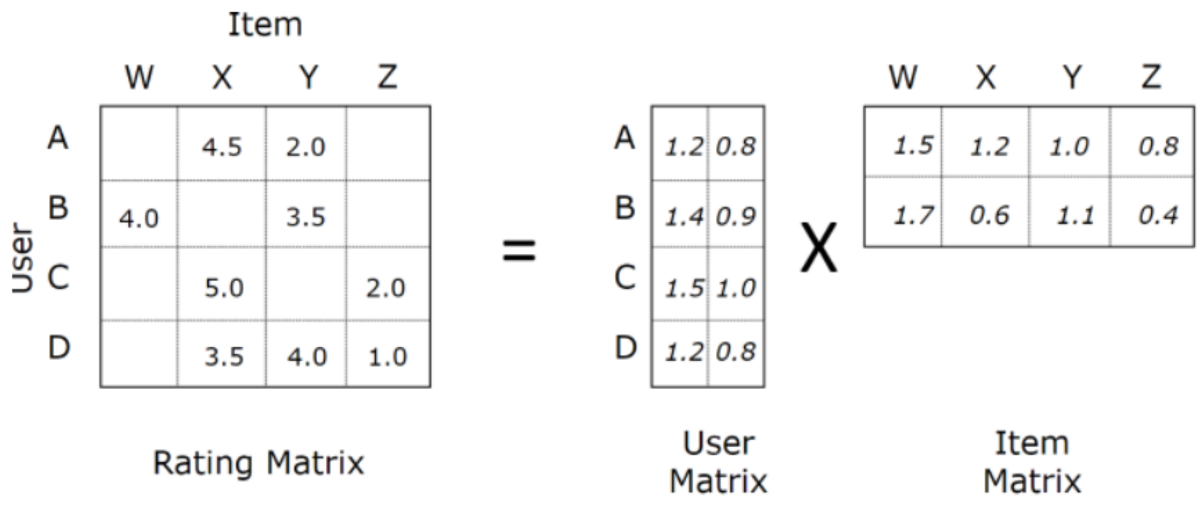
$$\mathrm{PC}_j = (a_{j1} \times \text{Predictor } 1) + (a_{j2} \times \text{Predictor } 2) + \cdots + (a_{jP} \times \text{Predictor } P).$$

Principal Component Analysis was fairly simple and straightforward so we gave it a try in implementing it within our code. We already had some code written for HW7 where we had applied PCA to market analysis, and our team figured it would be a simple transition to use our columns of predictor values rather than the market data. The pca() function in Matlab provides three sets of data, coeff, score, lambda, and using these we can create a more advanced weighted average.Using the latent values, we can find the % contribution of each predictor value column, and apply it as a weight to the predictor score. We then sum up the newly weighted predictor scores to get a single column of predictor scores from 0 to 100.

Originally we then applied a global threshold where anything that was above a 50 was a 1 and anything below was a 0, but this resulted in a fairly low score in the 0.7's. Instead, we discussed that it might be better to look at each user's scores individually. These scores can be used as an input to apply a varying threshold based upon the users given scores. Each user has 6 scores that can be used to create a unique threshold, in our case we took the average of these scores and said that anything above that users average score is a 1 and anything below is a 0.

Matrix Factorization:

Matrix factorization aims to find the relation between user and item. Matrix factorization algorithms work by decomposing the user-item interaction matrix into the product of two lower dimensionality rectangular matrices. By multiplying two lower dimensional matrices we will be able to fill the empty cells in the rating matrix. It give an idea about whether user A likes Z or how much it likes.



$$\begin{array}{c|cccc} & W & X & Y & Z \\ \hline A & & 4.5 & 2.0 & \\ B & 4.0 & & 3.5 & \\ C & & 5.0 & & 2.0 \\ D & & 3.5 & 4.0 & 1.0 \end{array} = \begin{array}{c|cc} A & 1.2 & 0.8 \\ B & 1.4 & 0.9 \\ C & 1.5 & 1.0 \\ D & 1.2 & 0.8 \end{array} \times \begin{array}{cccc} W & X & Y & Z \\ \hline 1.5 & 1.2 & 1.0 & 0.8 \\ 1.7 & 0.6 & 1.1 & 0.4 \end{array}$$

After we learned about matrix factorization, we wanted to apply this approach to our project. We used Google Colab to be applied to efficiently run the pyspark. First, we started SparkSession and imported our train and test data to Google Colab. We made some adjustments to our data frame before building an ALS model. ALS model is known as Alternating Least Squares matrix factorization. Imported our model from pyspark.ml.recommendation. Train our model with training data and transform the model by using our testing data, and we got the new prediction values

Logistic Regression:

Logistic Regression is a technique used to understand relationships between multiple variables and estimate the probability of it being 0 or 1. Generally logistic regression would follow a model like this: **logit(p) = a+ bX_1 + cX_2 (Equation **)**. While working on this part of the code, the team used Pyspark since it had Logistic Regression built in the following package: from pyspark.ml.classification import LogisticRegression. Using the beta coefficients allows for us to to show the expected change in log odds for outcome for unit change. The team was able to plot out the beta coefficient and this gave the team a better understanding of how important the beta coefficients are to the program. With this information, the team was now able to plot the False Positive Rate (y label) over True Positive Rates (x label). This would produce a ROC curve. With this the team can use the model that was built to make predictions. It was necessary to split up the data we had into training and test data. The train data was used for the model and then once the model was built the test data would be used. Using a logistic regression model the team could now predict 120,000 values present as either 0 or 1 based on the model that was produced.

Decision Tree:

Decision Trees are used since they are easy to interpret, handle categorical features very well, extend to multi-class classification, do not need feature scaling, and are able to capture non-linearities and feature interactions. Using the following formulas:

$$Entropy = -\sum_{i=1}^{n} p_i * \log(p_i)$$

$$Gini\,index = 1 - \sum_{i=1}^{n} p_i^2$$

Information Gain

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N} I(D_{left}) - \frac{N_{right}}{N} I(D_{right})$$

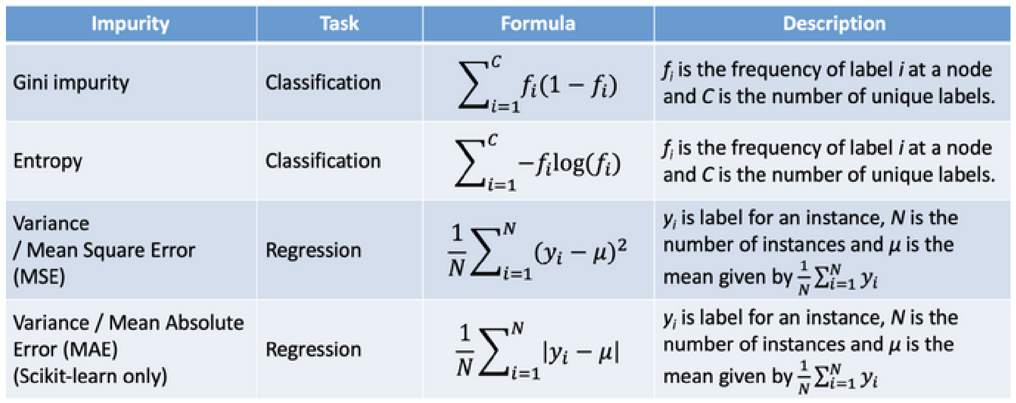
f: feature split on
D_p: dataset of the parent node
D_{left}: dataset of the left child node
D_{right}: dataset of the right child node
I: impurity criterion (Gini Index or Entropy)
N: total number of samples
N_{left}: number of samples at left child node
N_{right}: number of samples at right child node

It is possible to calculate the decision trees. Lucky for the team, the math was used by the following package in Python: **from pyspark.ml.classification import**

DecisionTreeClassifier. It was necessary to split up the data we had into training and test data. The train data was used for the model and then once the model was built the test data would be used. Following steps from lecture and online resources, the team was able to make the model for Decision Tree for our data and from there the team used it to predict 120,000 data worth of 0 or 1 for respected TrackID.

Random Forest:

Random Forest is used due to its ability to perform both regression and classification tasks. Random forests produce good predictions which are easy to understand and can handle large datasets very well. Also, the level of accuracy tends to be higher than the decision tree. With around 120,000 data samples, it made sense for the team to test out this form of classification/regression. Python has a package called: **from pyspark.ml.classification import RandomForestClassifier.** The math used by random forest are complex, but overview of the formulas are provided here:



Impurity	Task	Formula	Description
Gini impurity	Classification	$\sum_{i=1}^{C} f_i(1-f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels.
Entropy	Classification	$\sum_{i=1}^{C} -f_i\log(f_i)$	f_i is the frequency of label i at a node and C is the number of unique labels.
Variance / Mean Square Error (MSE)	Regression	$\frac{1}{N}\sum_{i=1}^{N}(y_i-\mu)^2$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{N} y_i$
Variance / Mean Absolute Error (MAE) (Scikit-learn only)	Regression	$\frac{1}{N}\sum_{i=1}^{N}\lvert y_i-\mu\rvert$	y_i is label for an instance, N is the number of instances and μ is the mean given by $\frac{1}{N}\sum_{i=1}^{N} y_i$

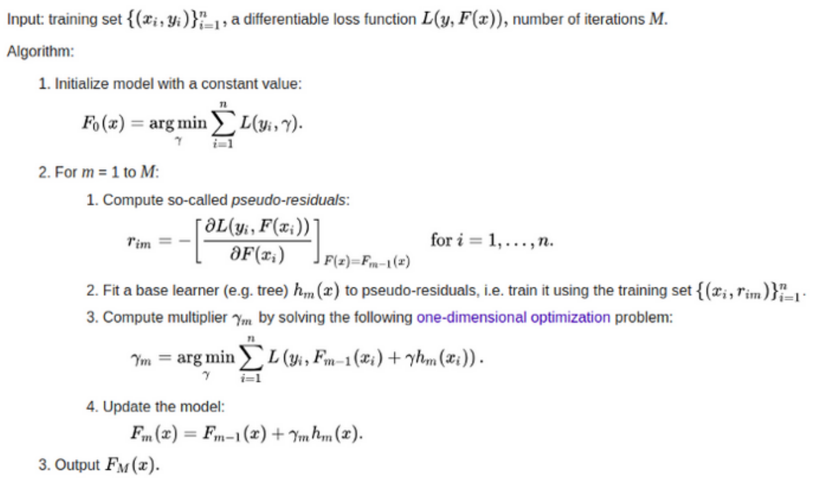
$$ni_j = w_j C_j - w_{left(j)} C_{left(j)} - w_{right(j)} C_{right(j)}$$

- ni sub(j) = the importance of node j
- w sub(j) = weighted number of samples reaching node j
- C sub(j) = the impurity value of node j
- left(j) = child node from left split on node j
- right(j) = child node from right split on node j

Very similar to the other classification tools, the data need to be split into train and test sets. Once that was done, using a package from Python called for the team to make a model using the train set and once the model was made the test set would be used for final predictions for 120,000 data worth of 0 or 1 for respected TrackID.

Gradient Boosted Tree:

This machine learning technique is used for both regression and classification tasks. It uses a prediction model in the form of an ensemble of weak prediction models. This model is good due to it being about to handle decreasing bias error. Also, gradient boosted trees use iterative functional gradient algorithms to minimize loss function to choose a function that points towards negative gradient. The package in Python that helped with building the model for Gradient Boosted Tree Classifier was from pyspark.ml.classification import GBTClassifier. Using this package allowed for the team to spend less time on complex math and more time with building a good gradient boosted tree model. The math formulas are the following:



Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

```
1. Initialize model with a constant value:
       $F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$
2. For m = 1 to $M$:
    1. Compute so-called pseudo-residuals:
           $r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}$ for $i = 1, \ldots, n.$
    2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
    3. Compute multiplier $\gamma_m$ by solving the following one-dimensional optimization problem:
           $\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$
    4. Update the model:
           $F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$
3. Output $F_M(x)$.
```

$$L = \left(y_i - y_p\right)^2$$

$$\frac{\partial L}{\partial y_p} = -2\left(y_i - y_p\right)$$

$$-\frac{\partial L}{\partial y_p} = 2\left(y_i - y_p\right)$$

$$where\ F\left(x_i\right) = y_p \qquad F_0(x) = \gamma_{optimal} = \min_{\gamma} \sum_{i=1}^{n} L\left(y_i, \gamma\right)$$

Very similar to the other classification tools, the data need to be split into train and test sets. Once that was done, using a package from Python called for the team to make a model using the train set and once the model was made the test set would be used for final predictions for 120,000 data worth of 0 or 1 for respected TrackID.

Ensemble Testing:

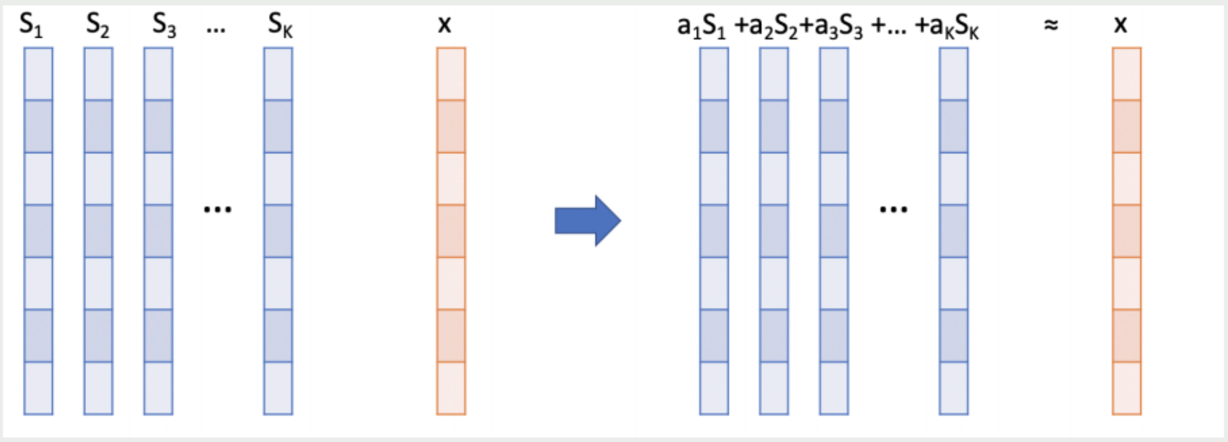
The ensemble method's primary goal is to combine multiple models to develop better results.

We gather all the submissions we have done in the Kaggle, convert zeros to -1, and combine them in one data frame. We also collected our Kaggle score for our previous submissions and multiplied it with the submission matrices. Next, each score is multiplied by its submission matrix. After that completed calculation for each submission vector and its score, we sum them up. Calculation is basically shown in the figure below.

a: Represents the scores

S: Represents the matrices

$$a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K \approx \mathbf{x}$$



We changed “0”s to “-1” because when the prediction is equal to the ground truth, we will get the result as “1” no matter if the prediction is “1” or “0” itself. When the prediction is not matching the ground truth, we will get the result as “-1”. After this calculation we calculated Least Square Solution aLS:

$$\mathbf{Sa} \approx \mathbf{x} \quad \Rightarrow \quad \arg\min_{a_1, a_2, \cdots, a_K} \|\mathbf{x} - (a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K)\|^2 = \arg\min_{\mathbf{a}} \|\mathbf{x} - \mathbf{Sa}\|^2$$

$$\arg\min_{\mathbf{a}} \|\mathbf{x} - \mathbf{Sa}\|^2 \quad \Rightarrow \quad \mathbf{a}_{\mathsf{LS}} = \left(\mathbf{S}^T\mathbf{S}\right)^{-1}\mathbf{S}^T\mathbf{x}$$

aLS tells us how to put weights on different solutions and generate a linearly combined new solution vector which should have the smallest distance to the ground truth vector x.

$$\mathbf{s}_{\mathsf{ensemble}} = a_1\mathbf{s}_1 + a_2\mathbf{s}_2 + \cdots + a_K\mathbf{s}_K = \mathbf{S} \cdot \mathbf{a}_{\mathsf{LS}} = \mathbf{S}\left(\mathbf{S}^T\mathbf{S}\right)^{-1}\mathbf{S}^T\mathbf{x}$$

This is the ensemble method.

2. The performance you observed and a brief explanation.

Simple Averages Score:

Our simple average score did not perform very well, we tried a couple of different submissions using a varying number of features, but the scores generally remained around the value of 0.75, which was not terrible, but not great. We did not expect the score to be great as it was just taking the average of the feature scores.

Weighted Averages Score:

The weighted average score did perform slightly better than the simple average score, getting an average score of around 0.78. We expected that the score would improve as we believed that a user's feature scores should not be equally weighted i.e. a lower rating on the artist score might be more important than a low score on a specific genre, or possibly vice-versa.

PCA Score:

We applied our PCA code to a couple of our pre-processed data.csv files and were able to get scores of 0.86259 and 0.86679 which were better than our other scores. We were glad that we were able to achieve these scores, but we believe that strictly relying on PCA will not get us the best results that are possible. If our data set was not linearly distributed then PCA values would not be the best to represent our data, in this case it seems like our data is linearly distributed, but it's not a good idea to always assume so.

Matrix Factorization Score:

Our group did not get a very good score for matrix factorization after we wrote a script for it. This may have been due to our code being wrong, or we may have possibly missed something, but the score was giving us below a 0.6, which was far below our expectations. We discussed it amongst ourselves as well as with other groups, but there was a shared expectation that the scores would be higher.

Logical Regression Score:

The logical regression score came out to be 0.84507 which was higher than most other scores of the 4 classifiers that we applied to the data, this one performed the best, being slightly ahead of the decision tree classifier. The score was about what we expected, although we may have hoped it was higher, and we think it performed relatively well compared to previous scores.

Random Forest Score:

The random forest score came out to be 0.82108 which was tied for the lowest score of the 4 classifiers that was applied to the team's data. The score was lower that what the team expected, but even with more time it seems like accuracy could not have been boosted.

Gradient Boosted Tree Score:

The gradient boosted tree score came out to be 0.84398, just behind the logical regression score. Similar to the other classifiers, we hoped that the score would be higher, but it still performed relatively well compared to previous submissions.

Decision Tree Score:

Decision tree score came out to be 0.82108 which was tied for the lowest score of the 4 classifiers that was applied to the team's data. The score was lower than what the team expected and after trial and error, the team could not increase said score.

Ensemble Testing Score:

After assembling our best submissions, we applied the ensembling method to all of the previous submissions. We were able to achieve a score of 0.85563 which was better than all of our other scores, except for the PCA weighted scores. As a group we thought that we might see more of an improvement, but we believed that some of the lower score files might have been dragging the score down. After removing some of the lower score files and trying some newer methods that generated higher scoring files, we achieved the score as mentioned above, which we were somewhat happy with as it was an improvement to the score.

3. The overall results after ensembling and your comments.

After a ton of back and forth on hand selecting data for ensembling, the team decided to use the best 11 scores ranging from PCA, Gradient Boosted Tree, Logical Regression, weighted average, and random forest. These 11 scores were used for the ensemble method due to their high accuracy present and it contained most of the methodologies taught in class. The scores from the Kaggle were manually added as an array of Kaggle scores. The team combined all of the values with User_ID, predictor + (num for each instance) into one file and wrote it into a new file called: all_data_11_values.csv. From there using the math that was taught in class, the team performed ensemble math required. Once that was done, the team had to get the final prediction scores and write them to a new file called Ensemble_Predictions_Test_11_values.csv. The accuracy of said file was around

0.85563 which was one of the highest results the team was able to achieve. The result from other testing with ensembling seemed to get better with the more high scoring csvs the team had, but as the deadline for the project was approaching the team had to stop experimenting and had to work on the report.

4. Attach the script codes in separate files:

A folder called **Code** will contain all of the code used by the team for this semester long project.

PCA_WeightedAvg.m: Takes the 82_features.csv file and applies the pca() function to the columns of feature data, then outputs the 0-100 predictor value for each track_id per user.

PCA_WeightedAvg_RecommendByUser.m: Takes the output.csv file of the PCA_WeightedAvg.m and applies a varying threshold to the scores based on each user's individual scores.

82_features.csv: A csv file with all of the UserId's and TrackId's plus all of the feature scores for each user

Weighted_average.py: Takes Our original formatted data that has a weighted average score based on the user preference. This code was originally written with a weighting of 50/50 for the simple average, but later we adapted it for weighted averages. This file will have to be run through threshold code

Weighted_average_with_8_features.py: Takes the 82_features.csv and applies a set weighting to the first two predictor columns, this well then be run through threshold code

logistic_regression.py: Logistic regression uses the final project test data and the data is prepared beforehand and then fit into whatever classifier we apply it to in order to get a resulting file for submission.

decision_tree.py: Decision tree uses the final project test data and the data is prepared beforehand and then fit into whatever classifier we apply it to in order to get a resulting file for submission.

random_forest_classifier.py: Random forest uses the final project test data and the data is prepared beforehand and then fit into whatever classifier we apply it to in order to get a resulting file for submission.

gradient_boosted_tree_classifer.py: Gradient boosted tree uses the final project test data and the data is prepared beforehand and then fit into whatever classifier we apply it to in order to get a resulting file for submission.

matrix_Factorization_Music Recommendation.ipynb: This code was written using pyspark, and adapted from the code given to us in class. We read in the trainItem.data, grab its columns of data, create an ALS model, then fit the data to the model. We generate a predictions file that we can then threshold and submit

hw10.py: used for ensembling method to take x amount of good files with a amount of Kaggle score and produces a ensemble csv file