Classifying NBA Stats with Python

Gene Zaleski
Intro to Data Mining
Amal El Gehani

Rowan University
zaleskig8@students.rowan.edu

***Abstract*— This document outlines the challenges and work done on my final data mining project for which I scraped all player stats from nba history to make predictions about entered stat lines.**

I. Introduction

For my final project, I used a python library called nba_py to scrape stats.nba.com to acquire all stats from every player listed from all nba seasons dating back to 1946. Once this was accomplished, I converted all the data to easy to read lists represented by a csv file. Finally, I used the list compiled in my csv file to run both naïve bayes and knn classification on the data set to make predictions about entered statistics. To evaluate these methods, I found the error rate of the results and timed the execution of my classifiers. Application developed using Linux CentOS 7, and a mixture of python 2.7 and 3.7

II. Accuiring Data

A. *Using nba_py*

To collect my data, I used a python library made specifically for collecting data from stats.nba.com. This library is called nba_py, and can be found at https://github.com/seemethere/nba_py. This module contains functionality to get information about teams, players, individual games, and for web usage daily scoreboards. For my project, I made use of the nba.Player module. This allowed me to pull stats from nba.com player by player.

B. *Setting up nba_py and collecting data*

Consistent with all other modern websites, nba.com is always changing it's infrastructure to stay up to date with current technological trends. So, when using nba_py I had to make a few changes to the source code to be able to make requests from stats.nba.com. After successfully following the install instructions from the github page, making a request using nba_py would hang up my system. After doing some research, I realized I had to change the header information listed in nba_py's __init__.py file to include up to date addresses for different web browsers. The changes read as follows:

```
HEADERS = {'user-agent': ('Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/57.0.2987.133 Safari/537.36'),
    'Dnt': ('1'),
    'Referer': 'https://github.com',
    'Accept-Encoding': ('gzip,deflate,sdch'),
    'Accept-Language': ('en'),
    'origin': ('http://stats.nba.com')}
```

By updating these header references, the curl requests made by nba_py can successfully be made to stats.nba.com. Another challenge presented with using the nba_py module is that nba.com doesn't want bots scraping data from their site. This presented an interesting problem for me, as I had to put a sleep timer in between each request as to avoid being IP banned as a bot. A three second timer between each request allowed me to bypass this. However, it somewhat stymied my progress when getting stats for every nba player, because there are around 3,000 unique players in the history of the league. To get each player, first I went through the full league roster dating back to 1946, and collected all unique player IDs. Once I had a complete list of IDs, I removed any duplicate ID's from the list as many players have played more than one NBA season, and the duplicate IDs would lead to a large number of duplicate stats requests. This cut the stats collection time down an incredible amount. With all the player IDs collected, I could now collect career stats for every unique player. I did this by passing the player IDs to the nba_py function

```
player.getPlayerSummary(ID).headline_stats()
```

and writing the resultant statline to the text file "allPlayerStats.txt", which can be found in the project repository. The module was very effective and gave me all the information I needed to create a comprehensive data set which I could use to make predictions.

III. Data Preprocessing

To be able to effectively read and predict from the web-scraped data, it needed to be preprocessed. The pre-processing was fairly straightforward. It can be broken down into two main steps: Converting the retrieved and stored text file back from text into json, and writing that data into a csv file to be easily readable by my classifiers.

A. *Converting from text to JSON*

When I collected all player stats, I wrote them into a text file to be stored. Writing this straight to a csv file would make the collecting of stats process more convoluted, so I decided to take the simpler approach and write the retrieved data to a text file and convert it after. The retrieved data was in json format, however when I reviewed it there were extra characters that needed to be removed, such as extraneous 'u's and extra brackets. Once removed, I could use the python library *ast* to convert the processed strings back to usable JSON, and access the points, rebounds, and assists as key and value pairs. In hindsight, I think I would immediately write the retrieved data directly to my csv file, however I was focused on just retrieving the data making sure I created a comprehensive dataset, and I think doing it one step at a time was helpful to keep a steady workflow for the project.

B. *Writing data to CSV*

Once the retrieved statistics have been converted back to JSON format, I could easily write the data to a csv file. This was a very straightforward process as python's *csv* module makes it extremely easy to create a row in a csv file and subsequently write that row. For my dataset, I would write rows as points-rebounds-assists-class, where points, rebounds, and assists are standard floating-point decimals, and the class is a string detailing what kind of player that the row in question details. I looked online to determine what some standard boundaries are for all-star players and above average scorer, rebounder, and playmakers are, then used these boundaries to assign a class for each stat line. These boundaries can be easily changed to prioritize certain aspects of the game to classify in unique ways. All data preprocessing steps can be found in the file "convertStatsToCSV.py" in the project repository. *Note: "convertStatsToCSV.py" runs on python 3.7.*

IV. Classification with Naïve Bayes and Knn

To create a predictive model in my project, I used two types of classifiers: Naïve Bayes, and K-nearest neighbors. I used these classifiers to take a stat line and show what kind of player that the line belong to.

A. *Naïve Bayes Classifier*

The first type of classifier I used in my project was a Naïve Bayes classifier. In summation, Naïve Bayes iterates Bayes theorem over many rows in a dataset to find the distance between a new value and present values, and uses this to classify the new value. It is a simple, fast, and accurate algorithm. In my testing it was the more effective algorithm in classifying new rows as it had a smaller error rate (see V. Evaluation Methods). Creating the classifier was fairly simple, however I did encounter some road blocks to getting the classifier to work correctly. For example, I frequently encountered errors where the dataset I used would include some empty strings, (something I missed reviewing the collected data as there were thousands of entries) so I would have to filter all rows coming into functions for which calculations occurred so no errors when attempting to compare types would occur. Once the datasets were filtered, the classifier worked as intended. Initially, the results seemed off, however I just had to fine tune the way I was classifying player stats when creating my datasets. I did this to train the naïve bayes classifier in the most optimal fashion.

B. *K-nearest neighbors classification*

The second way I attempted to create a predictive model in my project was by using K-nearest neighbor classification. In summation, K-nearest neighbor finds the *k* closest entries in the dataset to the value in question. The classifier then finds the most frequent class ID in those neighbors and assigns it to the new value. This classifier was straightforward to implement, but produced similar issues to what I encountered while attempting to implement the naïve bayes classifier. The issues were fixed in a similar fashion, and the classifier works as intended. However, the results of KNN classification were not as accurate as naïve bayes, which will be talked about more in detail in the following section, *V. Evaluation Methods.*

V. Evaluation Methods

To evaluate the performance of my classifier, I created functions to measure the error rate of the classifiers and the speed of the algorithms. To evaluate the classifier's error rate, I created a function to compare the results of 10 runs of each classifier with 10 preset, correct runs, then printed the error percentage to be viewed. This was very useful in comparing the two classifiers, and showed me that in multiple tests the naïve bayes classifier would produce a 10 percent error rate, whereas the KNN classifier would produce a 30 percent error rate. Next, I evaluated the speed in which each algorithm took to complete. I did this by using python's *time* module to time the execution of both my algorithms, which I then compared. Naïve bayes is both faster and more accurate than KNN is.

VI. Conclusion

In conclusion, I created a system to classify an entered stat line using Naïve Bayes and KNN classification. I constructed a dataset of all unique nba players to train and test my classifiers on. This provided comprehensive and accurate classification. To evaluate these methods, I tested for their error rate and timed the execution of both algorithms, and found that Naïve Bayes was both faster and more accurate. This project was developed using CentOS 7 via python 2.7 and 3.7.