Mitch Kinney
Elaine Mou
Tom Werner
ECE:5995:0002

Web Application Final Report

Project Description

PlayByPlay is a web application created to provide situational statistics about football decisions through analysis of previous games' data. The application uses MongoDB to query 12 seasons of NFL plays and calculate the average results of given situations and chosen plays. The web application uses an AngularJS interface, powered by the Spark server framework making queries on an indexed MongoDB database. The database contains information from advancedfootballanalytics.com, and has been through significant pre-processing to allow us to execute queries that can be useful.

Architectural Overview

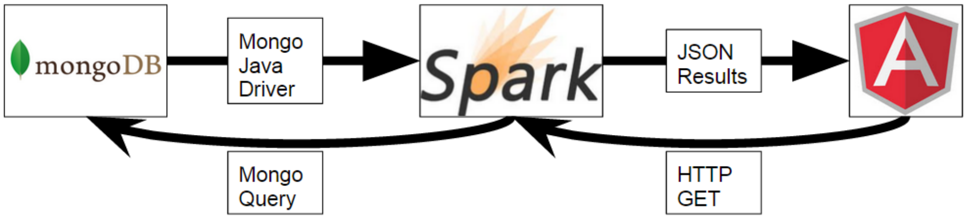
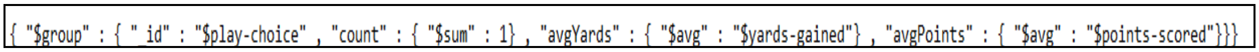


Fig. 1 - Architecture of PlayByPlay

PlayByPlay uses the AngularJS framework and HTML for its front-end interface, allowing users to specify a situation on the football field. This consists of which down the team is on, how many yards are left to go (out of 10), the range of yardlines the teams are at, and which team is on offense. Upon submitting the request, an HTTP GET request is sent to the Spark server, which queries the MongoDB database for statistics. The indexed database then finds average statistics on the relevant plays using the $aggregate, $avg, and $sum functions, and returns the results to the java server. At this point processing is done on the data to do things such as combining the statistics on complete and incomplete passes, left and right runs, and other things to increase the readability of the data. From here JSON data is passed back to the Angular frontend where it displays the results to the user as charts and tables.



```
{ "$group" : { "_id" : "$play-choice" , "count" : { "$sum" : 1} , "avgYards" : { "$avg" : "$yards-gained"} , "avgPoints" : { "$avg" : "$points-scored"}}}
```

Fig. 2 - Example query of database

The provided results include both the breakdown of play results for the given situation (e.g. what fraction of teams elected to pass or run), as well as the average points scored and yards gained for each play choice. Thus, the user can not only determine which decisions were most popular, but which ones were most effective. In addition, they can compare different teams' results to understand if their favored team is better or worse than average on certain plays.

Database Schema

Field	Type	Example
"gameid"	String	"20020908_SD@CIN",
"qtr"	Integer	1,
"min"	Integer	46,
"sec"	Integer	23,
"off"	String	"SD",
"def"	String	"CIN",
"down"	Integer	2,
"togo"	Integer	1,
"ydline"	Integer	1,
"description"	String	"(1:23) D.Brees pass to J.Norman for 1 yard TOUCHDOWN.",
"offscore"	Integer	3,
"defscore"	Integer	0,
"season"	Integer	2002,
"play-choice"	String	"PASS",
"yards-gained"	Integer	1,
"points-scored"	Integer	6

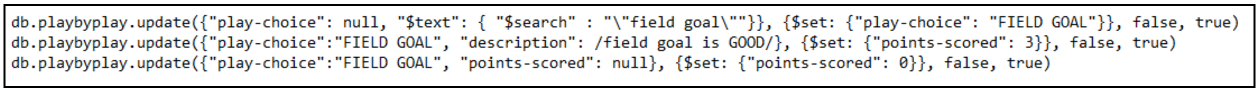
Fig. 3 - Database scheme with example record

The database schema is taken almost directly from the format of the original data with three additional columns added for query speed: "play-choice", "yards-gained", and "points-scored." In MongoDB each document is a play. We utilized the "off", "down", "togo", and "ydline" columns from the original data to construct indexes, and created our three additional columns from the text index on "description".

Data Statistics/ Indices Used

The dataset used for this project was originally compiled by a self-described football fanatic. The data collection we chose to use was an aggregation of every play from the past 12 nfl seasons with information on what game the play was run in, when in the game the play was run, where on the field the play was run, the team that ran it and defended against it, the score when the play was run, the down and to-go when the play was run, and most importantly a brief description of the play and the outcome. Each season was split into a different .csv file and each row within the file was a play. The total size of the data was 250 mb.

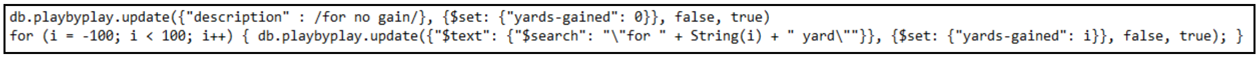
The data was loaded into MongoDB where we created an inverted index on the description field. It quickly became evident however, that using a text index for queries wasn't feasible for this project. Because of the content of the description field, which contains things like player names, play choices, and yard gains, the inverted index was almost as large as the actual dataset. Running queries with this index proved to be painfully slow, with simple queries taking over 5 seconds to execute. It was clear that significant pre-processing of the data was needed.



```
db.playbyplay.update({"play-choice": null, "$text": { "$search" : "\"field goal\""}}, {$set: {"play-choice": "FIELD GOAL"}}, false, true)
db.playbyplay.update({"play-choice":"FIELD GOAL", "description": /field goal is GOOD/}, {$set: {"points-scored": 3}}, false, true)
db.playbyplay.update({"play-choice":"FIELD GOAL", "points-scored": null}, {$set: {"points-scored": 0}}, false, true)
```

Fig. 4 - Example of MongoDB commands run for setting “points-scored”

The fields we added were play-choice, yards-gained, and points-scored. These were fields that were essential to our query results, but the information was contained in the description field. We then went through an iterative process where we tried to manually classify each play using text queries. Figure 4 is an example of the type of queries used when setting play-choice as well as points scored.



```
db.playbyplay.update({"description" : /for no gain/}, {$set: {"yards-gained": 0}}, false, true)
for (i = -100; i < 100; i++) { db.playbyplay.update({"$text": {"$search": "\"for " + String(i) + " yard\""}}, {$set: {"yards-gained": i}}, false, true); }
```

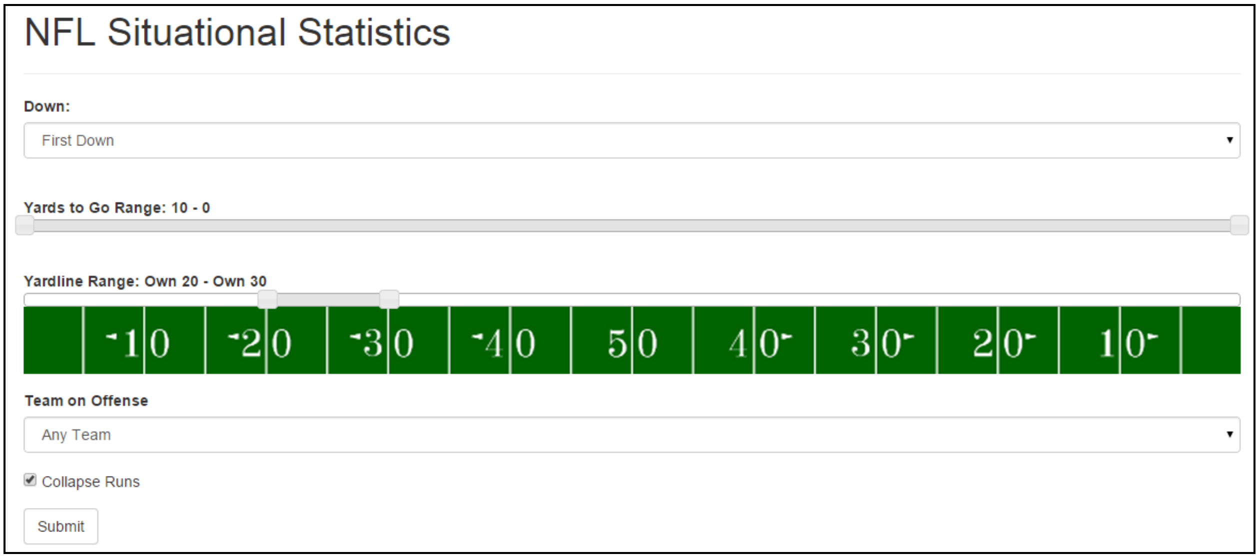
Fig. 5 - Example of MongoDB commands run for setting “yards-gained”

The yards-gained column was the most time-consuming field to update as our method was extremely inefficient. We ran a for-loop covering all the possible yards gained (i in [-100 to 100]) and searched for this number of yards in the text index of description (See Figure 4). If there was a number that matched i in the text index then we would update yards-gained, otherwise we would do nothing. This resulted in running over 200 * 473,900 records, which was inefficient but effective.

As a result of doing this preprocessing and our index on play type, we were able to speed up queries by over 15 times. A query that previously took 5.2 seconds to run now was executed in under .3 seconds. Without this preprocessing none of our queries would have been possible in a timely manner.

Features Designed and Implemented

The PlayByPlay web app allows the user to make comparisons about the effectiveness of different play choices as well as the historical breakdown of play choices. Since the demo presentation, many of the suggestions have been implemented. Below is a screenshot of the query interface, where the user can specify the field situation, as well as the level of reporting detail. The “Collapse Runs” selector allows the user to see more of the big picture play breakdown as opposed to the level of detail found in the data.



NFL Situational Statistics

Down:

First Down ▾

Yards to Go Range: 10 - 0

Yardline Range: Own 20 - Own 30

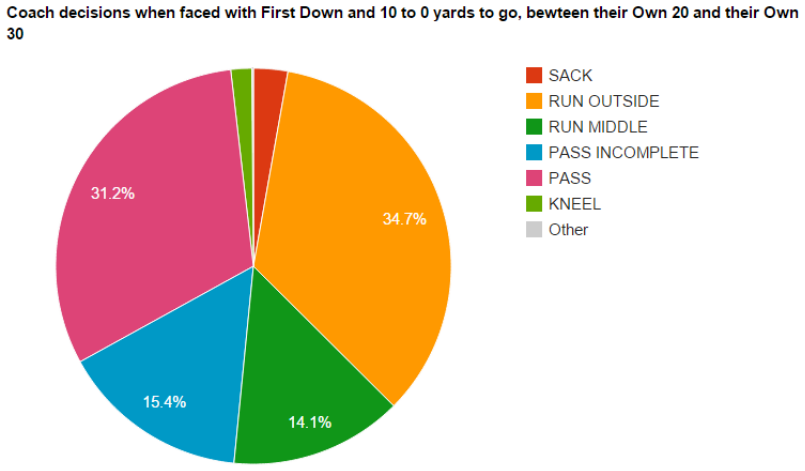
Team on Offense

Any Team ▾

☑ Collapse Runs

Fig. 5 - Screenshot of Situations options

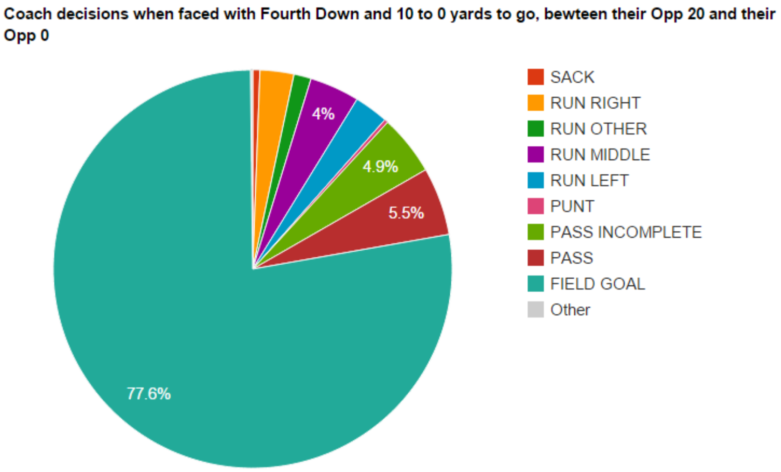
In addition, the method of reporting statistics has also been improved. Rather than forcing the user to switch between two different pages to see a comparison or breakdown, the tool now displays both. Below is the result of doing a “first play of the game” query, and the results.

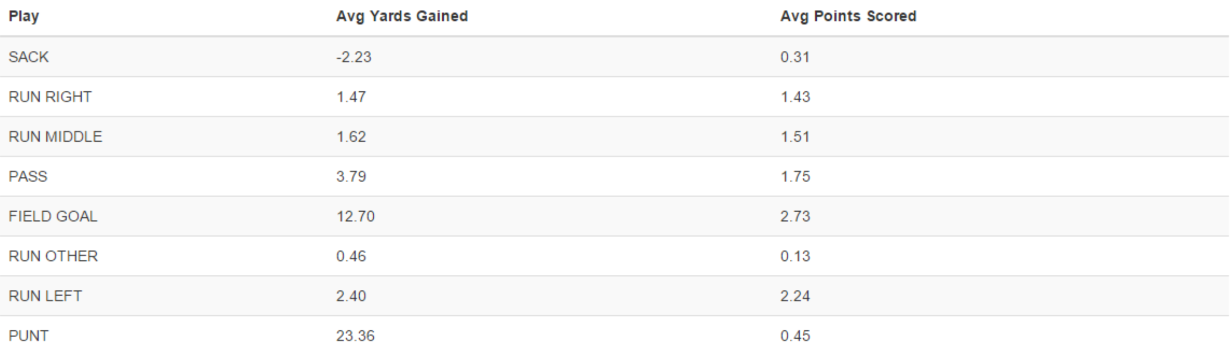


Play	Avg Yards Gained	Avg Points Scored
SACK	-5.86	0.04
RUN MIDDLE	4.34	0.03
SPIKE	1.64	0.00
RUN OUTSIDE	3.22	0.01
PASS	8.19	0.05
KNEEL	-1.03	0.00

We can see that an outside run is the most popular play, although if we look at the statistics below it isn't the most effective in terms of yards gained. In this new display system it is also easier to make a comparison between all of the play choices, rather than just two at a time.

There are also interesting results when the field position changes. When the ball is near the endzone (where teams score points), on fourth down (their last try to move the ball ten yards), running to the left is significantly more successful than running to the right.

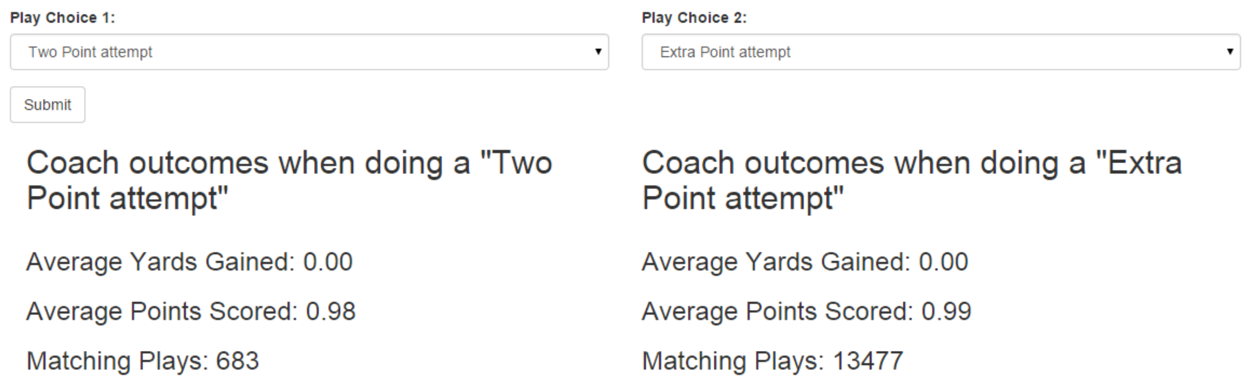

Coach decisions when faced with Fourth Down and 10 to 0 yards to go, bewteen their Opp 20 and their Opp 0



Play	Avg Yards Gained	Avg Points Scored
SACK	-2.23	0.31
RUN RIGHT	1.47	1.43
RUN MIDDLE	1.62	1.51
PASS	3.79	1.75
FIELD GOAL	12.70	2.73
RUN OTHER	0.46	0.13
RUN LEFT	2.40	2.24
PUNT	23.36	0.45

While the safe option of kicking a field goal is still the most popular, looking at the differences between average point scored on “run left” and “run right” is both interesting and potentially useful to a coach.

While now virtually obsolete after the post demo changes, the average yards gained and average points scored report was originally a separate query, used to compare two plays. The query interface was nearly identical to the situation query, but it allowed for the choice of two different plays to compare. Below is an example of comparing a two point conversion attempt (a risky attempt at two points) vs an extra point kick (a safe attempt at one point).



Play Choice 1: Two Point attempt

Play Choice 2: Extra Point attempt

Coach outcomes when doing a "Two Point attempt"

Average Yards Gained: 0.00

Average Points Scored: 0.98

Matching Plays: 683

Coach outcomes when doing a "Extra Point attempt"

Average Yards Gained: 0.00

Average Points Scored: 0.99

Matching Plays: 13477

Lessons Learned

When doing this project the lesson we learned was that pre-processing the data was especially important in getting efficient queries to run.

In the beginning stages of our project we noticed that our queries would take several seconds. This was because we were searching each description for the play we wanted to return, and checking the entire database multiple times. We then decided to create another nominal-valued column that would specifically contain play choice. Then our queries would simply return all documents matching a value in that column. The difference in run time was staggering. For example, simply running a regular expression search on the text-index of description for a pass play took about 5.2 seconds. With the pre-processing, running a search on the play-choice for “pass” took about .3 seconds, or a 17-fold increase in speed. This improvement was definitely worth the time and effort to clean and organize the data beforehand.

In addition, subtle errors in classifying plays lead to unrealistic statistics reports at first. Without domain specific knowledge, resolving these errors would have been nearly impossible. For example, one type of play is a “spike”, which is not a play that happens on first down. While testing, we discovered this play was happening on first down, because the MongoDB stemmer turned the player “T. Spikes” last name in to “spike”.

Other issues included finding far too high averages because new numerical columns had “null” values where they should have been “0”, causing only positive results to be considered. As a whole, generating new attributes was a delicate process, as the whole of nearly 500,000 records could not be closely inspected.

Open Issues/Future Work

The PlayByPlay web app is still a work in progress and in the future we would like to add on more features to increase usability and interest.

We would like to implement the ability to compare two team’s statistics. Fans of any sport enjoy being able to quantify how much better their sports team is than their rivals. Currently, a user would would have to make queries on two different pages and compare them manually. We aim to create a separate page where a user can select two different teams to compare and receive two different pie charts and expectation tables for each. Major differences could be reported by displaying the max difference in a section. For instance, if a team was found to only pass 10% of the time in a certain situation, while another team was found to pass 45% of the time, this would be reported to the user as a short sentence such as “The biggest contrast between these teams is the “ + “pass” + with a difference of “ + string(difference) + “.” .

We would also like to implement a k-nearest neighbor approach to plays such that small sample sizes provide reasonable statistics. For example, consider querying the play fourth and ten within your own one and ten yard line for the team the Dallas Cowboys. This is a rare play in football to start as it means the team was pinned back and then not able to advance the ball at all and the query is also narrowing it down to a single team. First we would make sure the number of matching plays returned is above a lower threshold. If not, we would expand the query out to all thirty-two teams. While there are definitely some team to team biases, we do not believe this bias would be as big as if we were to weaken the field position requirement. Then if the number of plays was still below this threshold, we would weaken the field position range. We would start by expanding out in both directions by five yards. In the case of our example, this would mean that the query would now look for plays that happened within your own one to fifteen yard line. Continue this until the threshold is met. We do not expect this to be so outrageous that the query might be taken over twenty yards away, as the minimum threshold of matching plays would likely be about 250 - not too difficult of a goal within a database of 473,900

We look forward to possibly continuing production of this app in the future and being able to implement these features in order to better both our app and our big data management skills.

Thank you for reading!

Installation Instructions

Unzip the project
Import the mongodump that was uploaded to dropbox (database: nfl, collection: playbyplay, teamSeasonVectors)
Run mongod
Using IntelliJ, open the project as a gradle project (choose settings.gradle). This will download all necessary dependencies.
Right click on StarterServer.java (src/main/java/com.bigdata.bigdatanfl.server) - you should see output from spark saying the server has initialized.
Go to localhost:4567/situations
The webapp should be ready to run.

To see all of the preprocessing commands, in the github project go to the /data folder, and open commands.txt

The main java mongo code is located in MongoImpl.java (com.bigdata.mongodb)

Other assorted screenshots are in the /screenshots folder

Additional Work Completed

I was interested in seeing if I could compute the similarity between different NFL seasons for teams using this data, with the hope I could find a sort of “coach signature”. I used teamSeasonVectorAggregate.js as the aggregation query to generate the teamSeasonVectors collection. The KMeans.java file allows the user to enter a team abbreviation in the command line and then it calculates the most similar team for each year. It does this with cosine similarity, where the vector is:

pass_percentage_0-20-1st_down
run_outside_percentage_0-20-1st_down
run_middle_percentage_0-20-1st_down
field_goal_percentage_0-20-1st_down
punt_percentage_0-20-1st_down

and so on for 20-40, 40-60, 60-80, 80-100, and for 1st, 2nd, 3rd, and 4th down (so around 200 total attributes).

Unfortunately, running this comparison doesn’t seem to give the coach signature I was hoping for. The results are sometimes interesting (2011 packers and 2007 patriots are similar, and were both teams that made it far in the playoffs), but overall the similarity measure wasn’t the most useful. My guess is that since it doesn’t take defense into the equation at all, it leaves a lot of information necessary out of the picture, and so there aren’t big differences between the teams.

I was also going to try doing a KMeans clustering on the vectors, but I didn’t have much luck trying to get Apache Mahout to work with the collection.