**Winnow Algorithm**

**Dataset:** Allstate Claims Severity

**URL:** <https://www.kaggle.com/c/allstate-claims-severity/data>

**Rows:** 188318

**Attributes:** 72 explanatory variables (categorical)

**Predictor:** Severity of Claim

**Aim:**

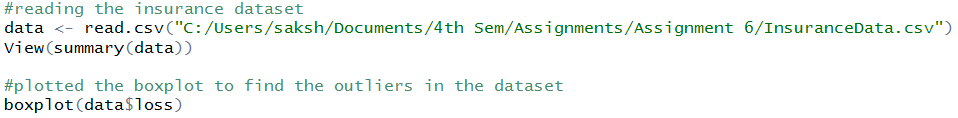
The aim is to predict claims severity for the Allstate Insurance company, to ensure a worry-free customer experience.

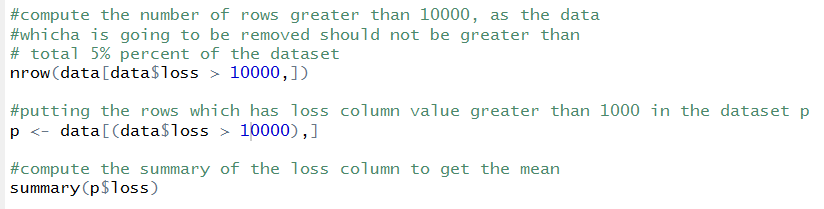
**Solution:**

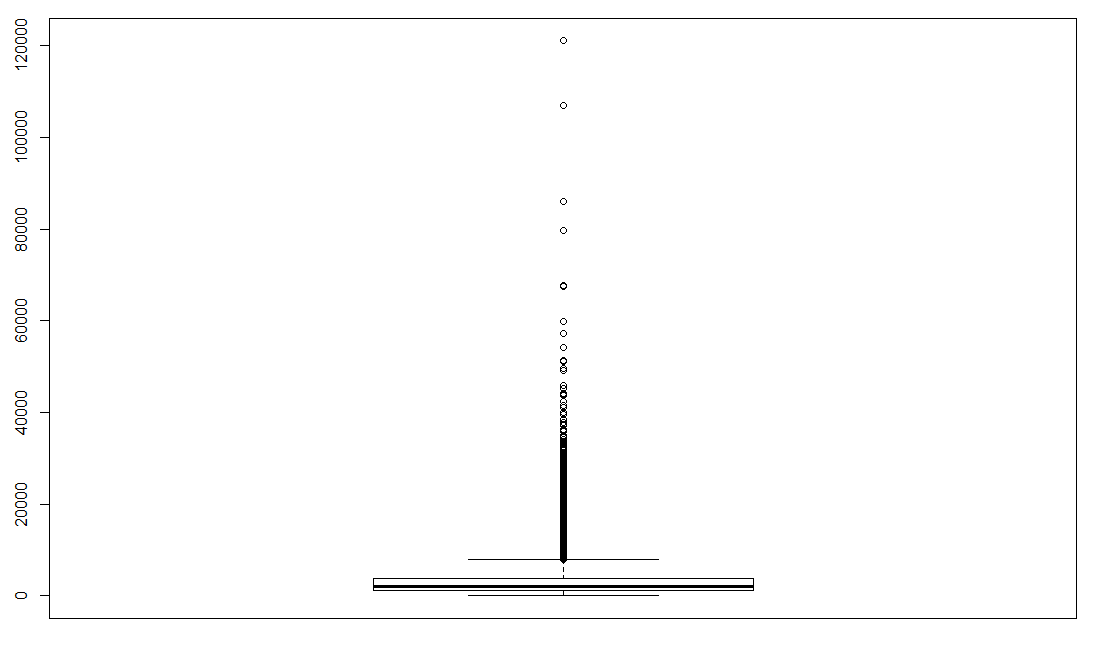
Since most of the explanatory variables are categorical, we can use the Winnow Algorithm. As per the Winnow Algorithm, train a binary classifier based on binary features, using a linear decision boundary. The prediction model assign weights to each future. To predict the state of an observation, check all the “**active**” features and sum up the weights assign to the feature. If the total is above a certain threshold, the result is true, otherwise it’s false. Winnow algorithm updates itself with learning process, it updates the weights to the feature when the model make mistakes. If the current model predicts the output correctly, don’t change anything. If it predicts true but should predict false, it is over-shooting, so weights that were used in the prediction (i.e. the weights attached to active features) are reduced i.e., divide the weights of every feature by alpha. Conversely, if the prediction is false but the correct result should be true, the active features are not used enough to reach the threshold, so they should be bumped up i.e., multiply the weights of every feature by alpha.

**Code Explanation:**

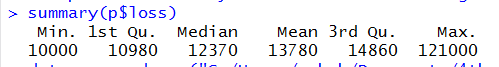
Allstate Insurance claim dataset has 72 categorical columns but the predictor column is continuous, I am changing it to binary by classify it if the value is greater than mean then 1 else it is 0. But for getting the accurate mean, first I have to remove the outliers which is not possible in C# for me, therefore I have used R to remove the outlier and compute the mean of the dataset and later used in C#.



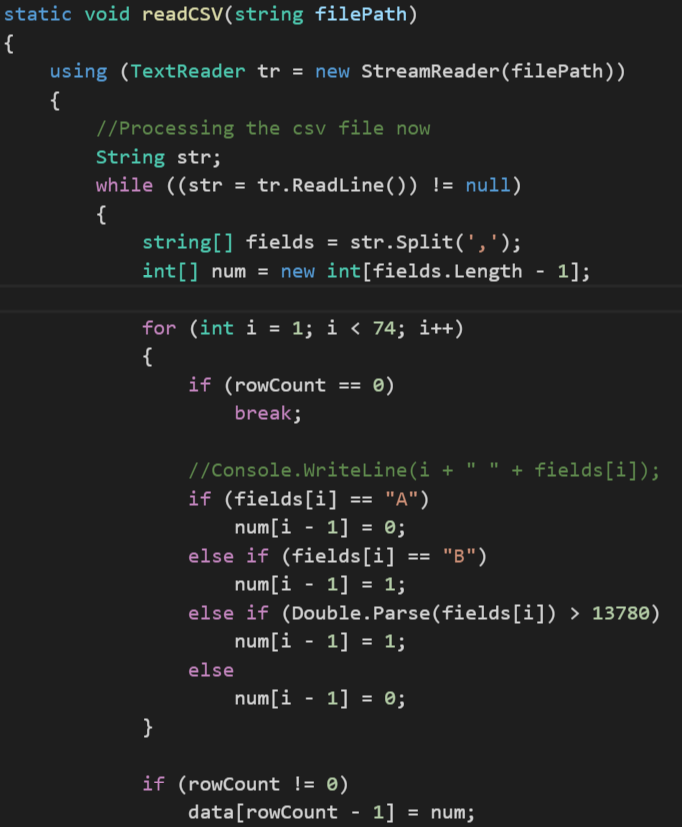




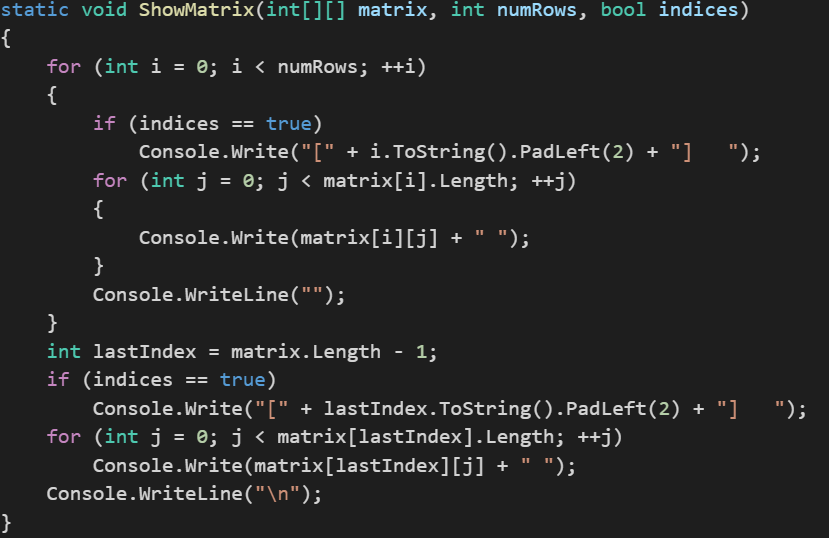
Mean of the loss column is, i.e., the mean of the severity of claim column is  **13780**



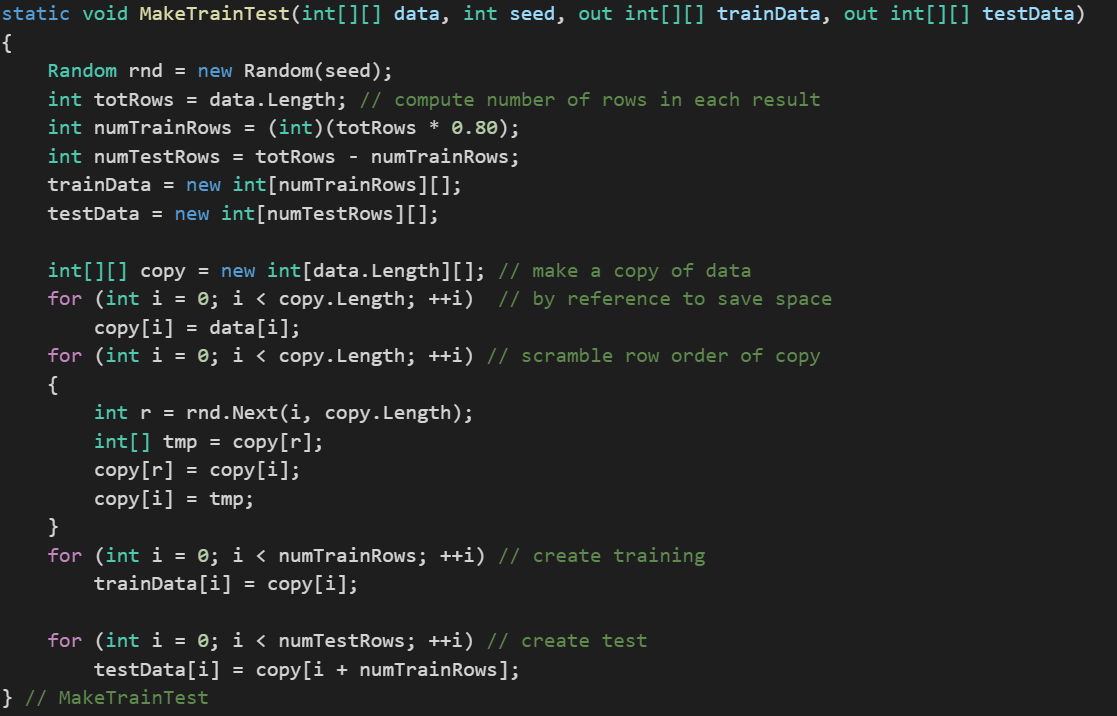
Now I have the **readCSV** function is to read the csv, and form a matrix, which is later used for training the model and later predicting the severity of the claim based on the observation or features given. Insurance dataset has categorical value as A and B, so for implementing the winnow, we have to convert those A and B into 0 and 1 respectively, apart from this the predictor colum is also a continuous variable, for converting that into binary, we are taking mean which is computed in R, and if the value is greater than mean than severity is 1, else it is not sever i.e., 0. We are running th for loop 73 times because there are 72 categorical variables, and starting the loop from 1, because 1st column is ID, which is not relevant to us.



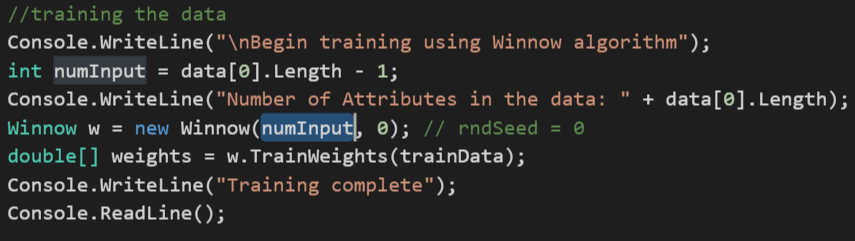
Once the dataset is formed, we can use **showMatrix function,** this function is to display first and last few rows of the dataset, it takes the dataset which we formed, the number of rows you want to display and boolean indices, do you want to show the index number of row or not.

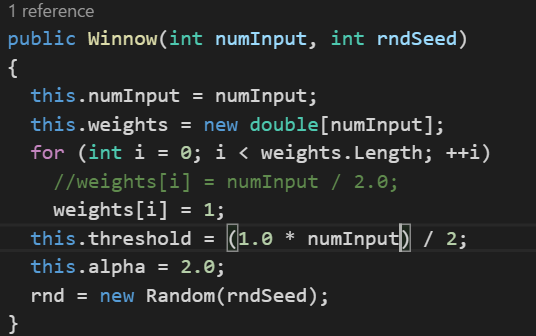


Once we have verified the dataset, we can now make the train and test from the dataset. For this we have **MakeTrainTest function,** this function make train and test data from the dataset, we split the data into 80% and 20%, 80% is to train the model based on Winnow Algorithm and rest 20% is to predict the accuracy of the algorithm. Before splitting the dataset, we set the seed value, seed value is a Random number which is used for generating the same trainData again. The function is going to output the result into **trainData** and **testData.**

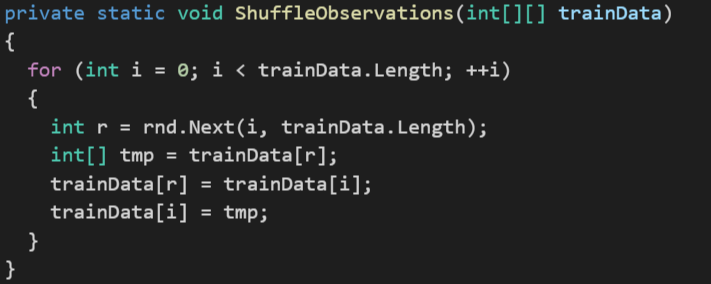


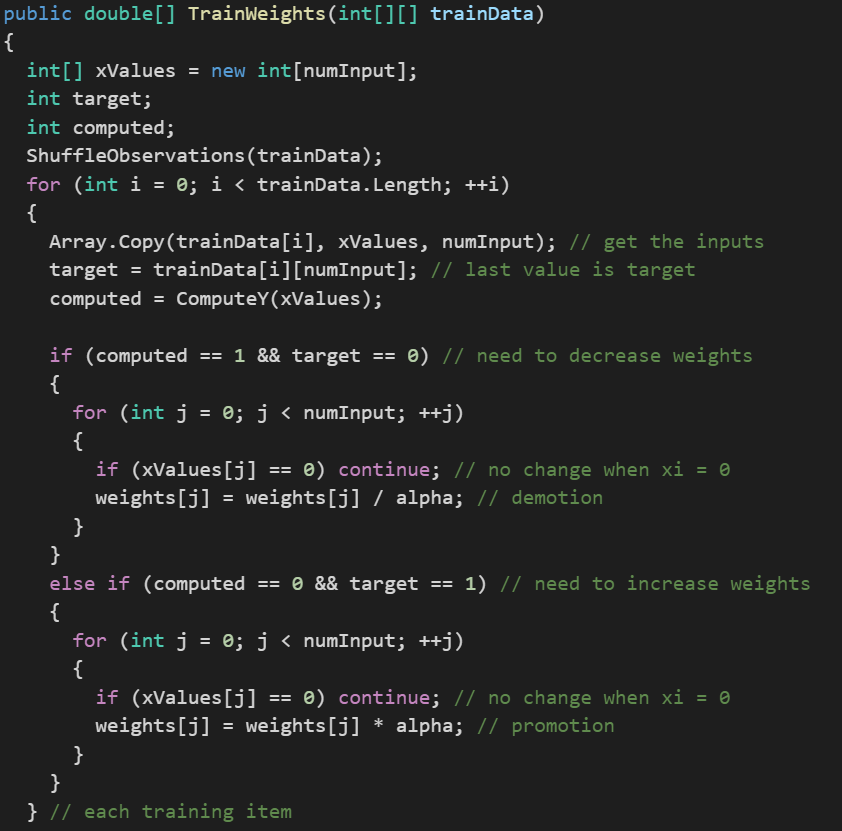
Once we have trainData and testData, we can now make the model by implementing Winnow Algorithm on the train data. We are going to initialize the winnow with the number of attributes/features, in our case they are 72 and then we call the **TrainWeights**, which is going to compute the weights of the feature. In the **Winnow class,** we have **assigned initial weights as 1** and **threshold as total features divide by 2** and **alpha value as 2**. Aplha is the value by which we are going to multiply or divide the weights.





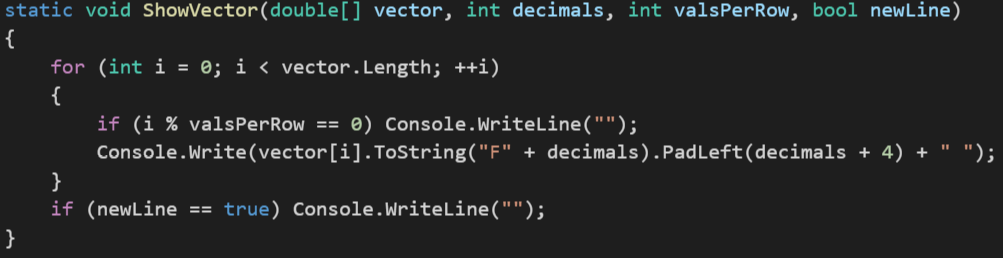
In the **TrainWeights function,** we are going to use the initial weights and then update the weights based on the conditions. If the current model predicts the output correctly, don’t change anything. If it predicts true but should predict false, it is over-shooting, so weights that were used in the prediction (i.e. the weights attached to active features) are reduced i.e., divide the weights of every feature by alpha. Conversely, if the prediction is false but the correct result should be true, the active features are not used enough to reach the threshold, so they should be bumped up i.e., multiply the weights of every feature by alpha. While training the data, we shuffle the observation as for computing the weights winnow use the train data. We are using the Fisher-Yates shuffle algorithm.



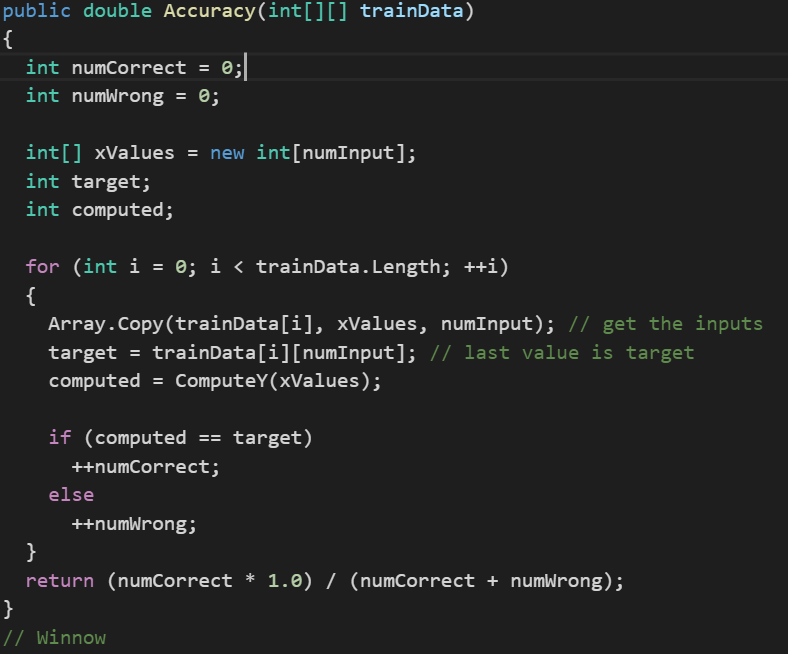


We have another **showVector function,** this function is used to display the final weights of the features, which is

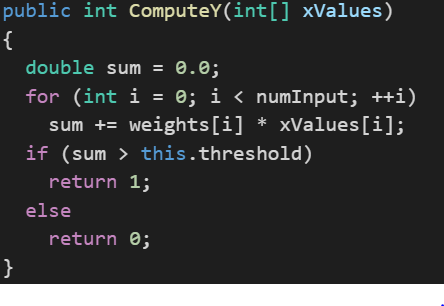
assigned to them by them by Winnow, based on the update function of winnow, it take number of features display in the row and whether you want everything to be in one line or more



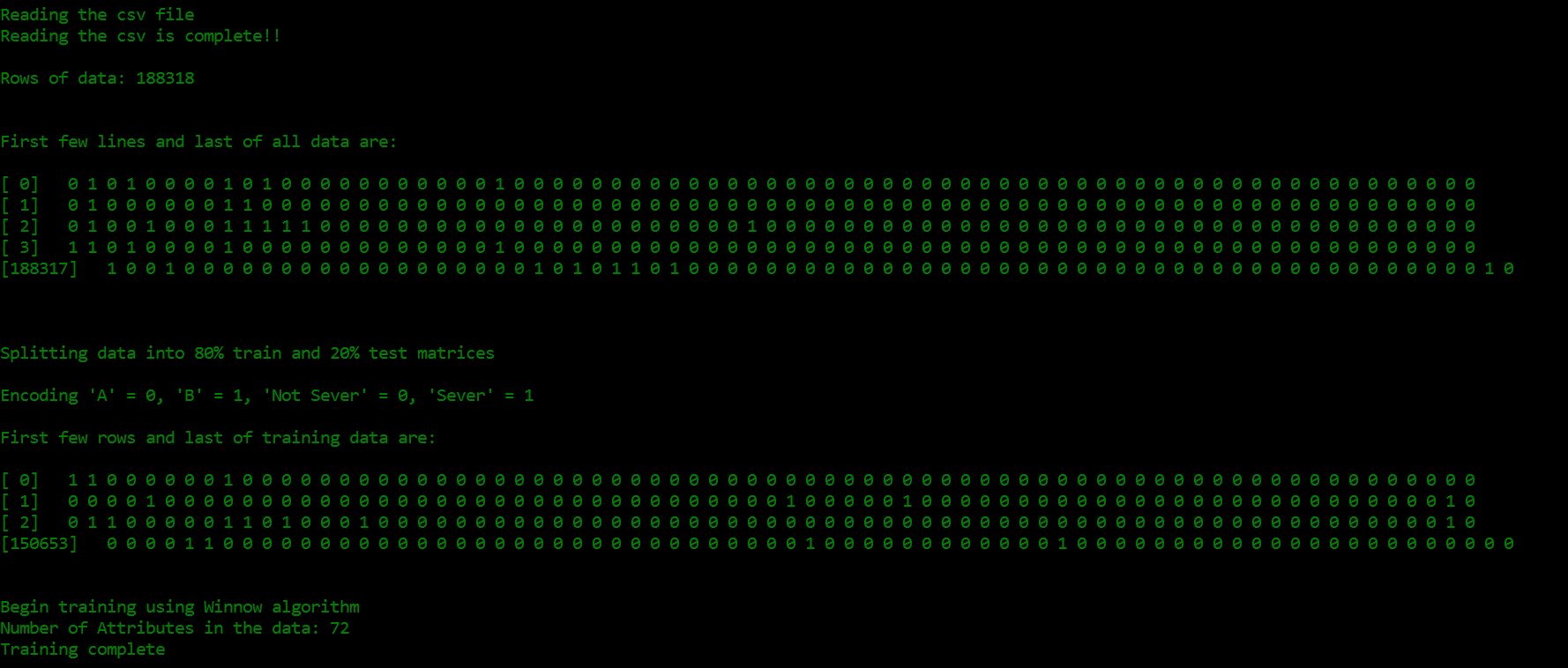
We can find the accuracy as well, after computing the weights of the features, this function computes the accuracy of the dataset, based on the weights which we have computed, what it does is it applies the algorithm to the dataset and try to predict the severity of the claim and then it compares computed value with the target value, later it gives the total correct divide by total number of observation.

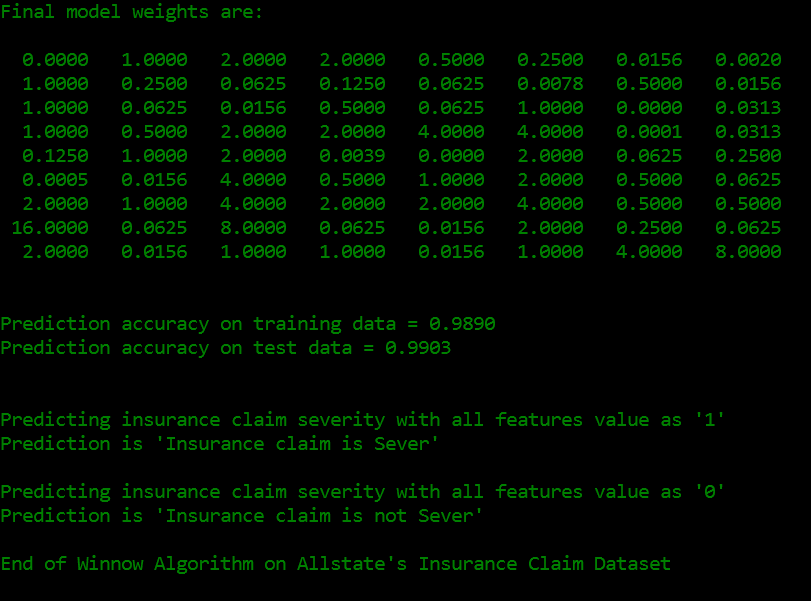


We have another **computeY function**, t his function is to predict the value of dependent variable, what it does is sum up the weights of all the feature whose value is 1 if sum is greater than threshold, it returns 1, else it return 0



**Output**





**My algorithm accuracy is 98.90% for train data and 99.03% for test data.**

1. **How could you improve on Winnow?**

**Solution:**

**For now, we are getting the best accuracy but to improve the accuracy of the Winnow Algorithm in cases** where we are not getting the appropriate result, we should shuffle the data**, we should come up a new random scrambling method** and run the TrainWeights functions i.e, **update function multiple times**, which will lead to better weights to the important feature and eventually that will lead to a better prediction. Winnow Algorithm is for linear function between dependent and independent variable in case of non-linear relationship between variables, we can introduce multiple hidden layers, this is concecpt behind Artifical Neural Network.

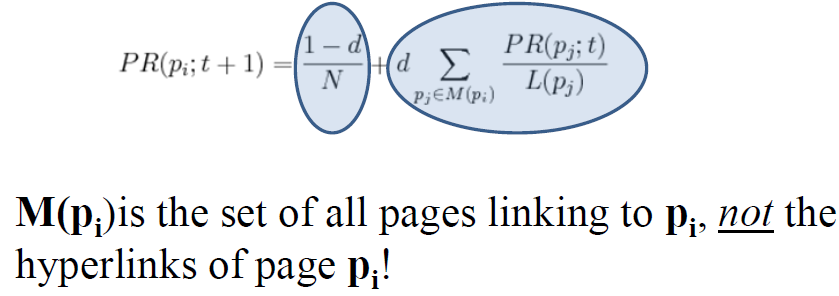
1. What’s the dataset size after which the winnow algorithm becomes unwidely?

**Solution:**

The dataset size for which winnow algorithm behaves unwidely is when datasets are small. It works with a very small loss in performance when a larger set is used, and in fact in some cases performance actually improves. The main performance loss is one of speed, but even this can be lessened, especially in the case of Weighted Majority, by a pruning method. Also Winnow gives better result when the number of features on which the weights depend are more. The dependent features gives better accuracy result. Winnow is much more robust in high dimension feature. Winnow perform better in cases when relationship is linear amony dependent and independent variables.

**XML PageRank Algorithm**

For page rank algorithm, we start with a number of web pages, where each page points to another web page. Each web page pointing towards a another web page is considered a vote for the another web page. More the votes for a web page, the higher the rank for the web page. Google came with a formula for calculating the web page



We are going to compute the Page Rank for a web page using map reduce. We are going to being with **(URL, list-of-URLs)** where list-of-URLs is the outlinks from the webpage. We are going to transform the above expression to **(URL, (PageRank, list-of-URLs))** where **PageRank is the initial guess, usually 0.5 or 0.1**.

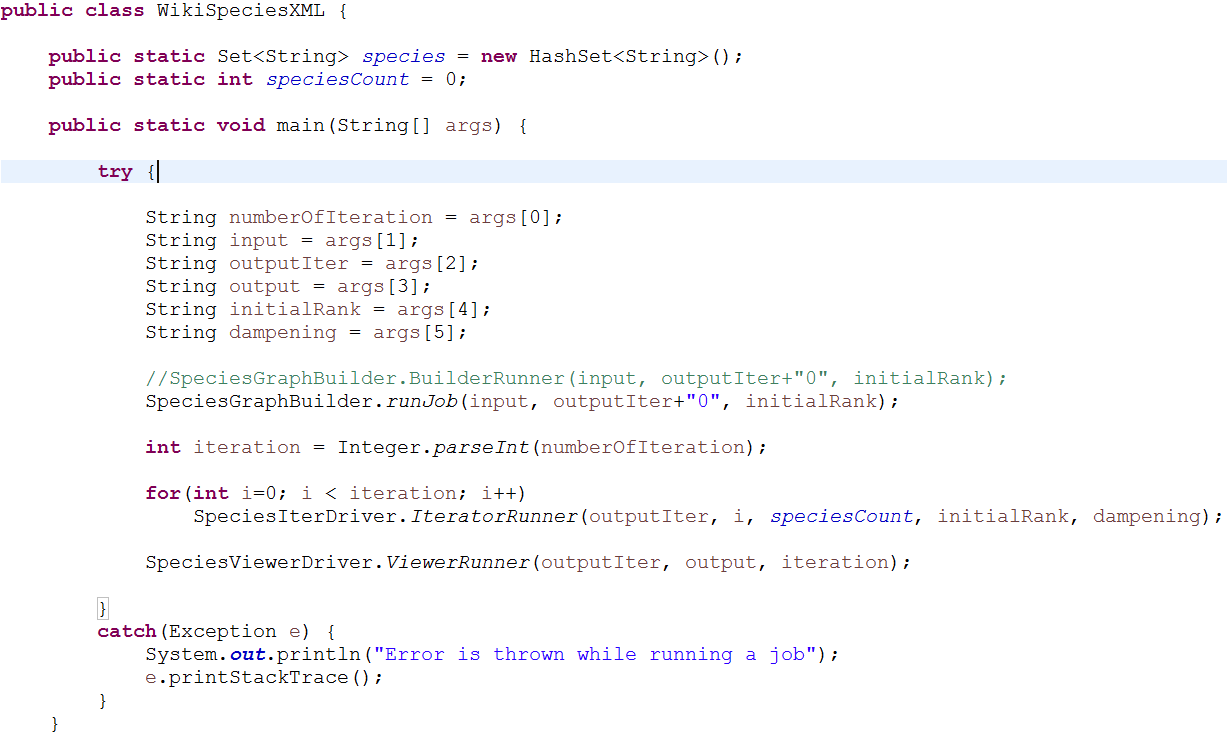
The above expression will act as input to the **Map**, the map for each ***u*** in the **list-of-URLs** will output **(u, URL, PR(URL)/Length of list-of-URLs)** i.e., the output link, with the voter, the page rank of the voter divided by number of output urls from the web page, the large the page rank of the URL, the more support to the web page will be give and we are going to output the **(URL, list-of-URLs)**.

Each Web page’s hyperlinks (outlinks, or votes) are “pivoted”, and we now tally up each Web page’s inlinks (voters) instead. In the **Reducer**, the input will be (URL, list-of-URLs) and many URLs pairs and it going to calculate the **(URL, (new PR, list-of-URLs))**. We are going to aggregate the votes from each voter and then going to multiply the aggregated value with the dampening factor. We are also going to add the the probability of the direc typing the URL and reaching the web page. The sum of the above expression will be the page rank of the web page. We are going to iterate through this mapper and reducer part couple of times and will observe that page rank is converging, after a while page rank will be constant and that will be the final page rank of the web page. In this assignment, we are implementing the Page Rank Algorithm on the XML input file instead of text file.

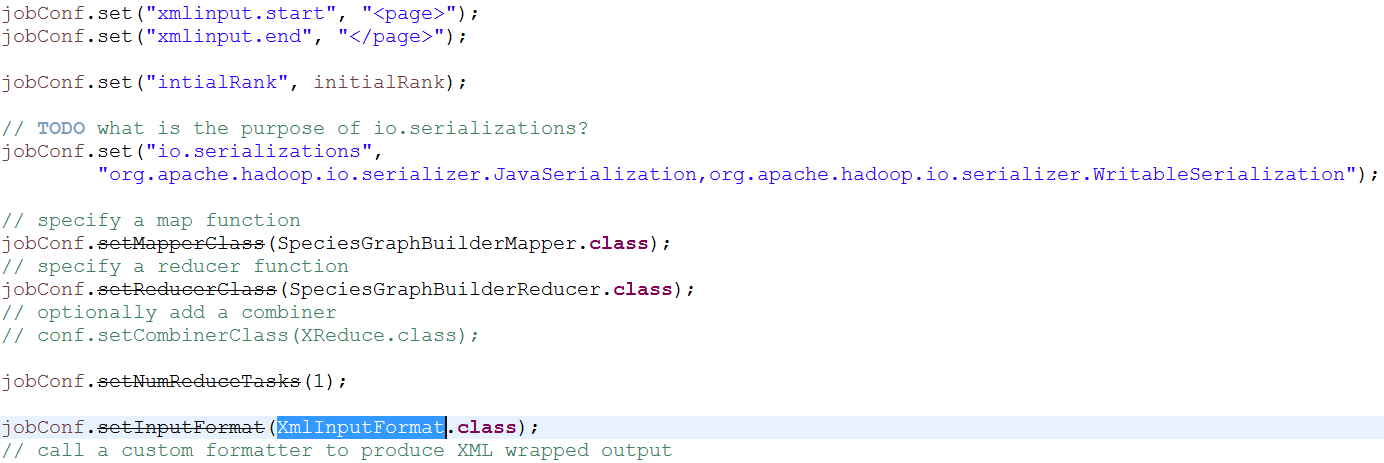
**Code Explanation**

For implementing the PageRank Algorithm, we have the concept it three parts which are named as **Graph Builder, Iterator and Viewer**. We have the main class name **WikiSpeciesXML**, which is going to call the Graph Builder, then count the number of total species from the output of the Graph Builder and then going to call the Species Iterator for the number of times the iteration is defined and the lastly Species Viewer, in which the result is present of the page rank.

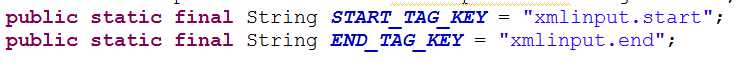
In **WikiSpecies**, we have parameterized the **Initial PageRank, Dampening factor, Number of Iterations, input directory path and output directory.** The main concept behind parameterizing the parameter is that it allows users to change them from the argument list without going within the code and changing them explicitly.



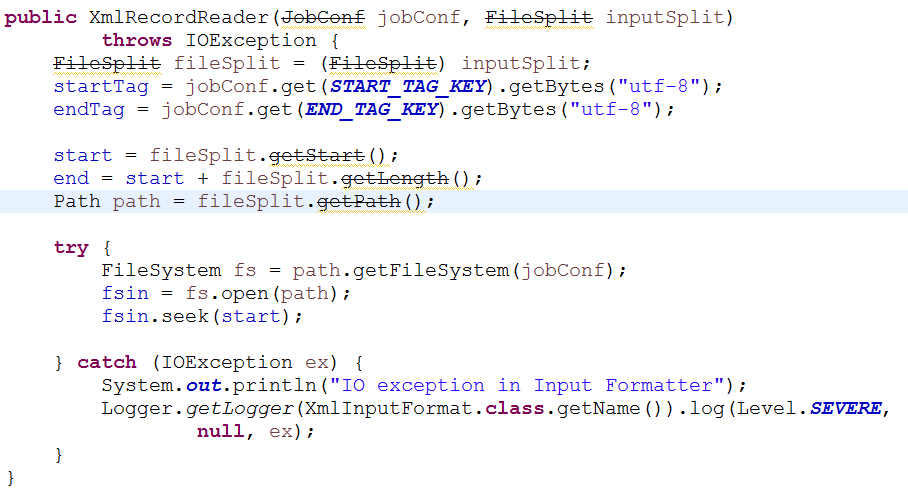
In **GraphBuilder,** since this time it is the XML file we are going to change the job configuration settings. We are going to set the “**xmlinput.start**” tag to “**<page>**” and “**xmlinput.end**” tag to “**</page>**” and we are setting the InputFormatClass as “**XmlInputFormat.class**”, which we have written on how and what to read from the XML file.



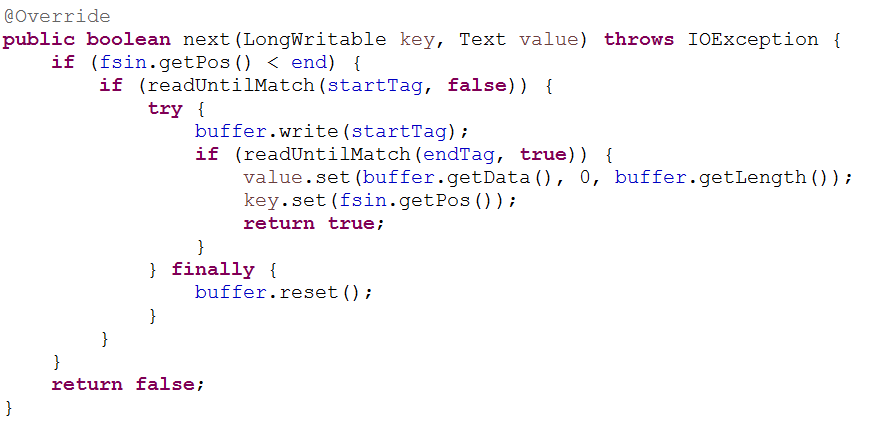
In **XMLInputFormat class**, we set the Start tag key and End tag key, which we are getting from the job configuration.

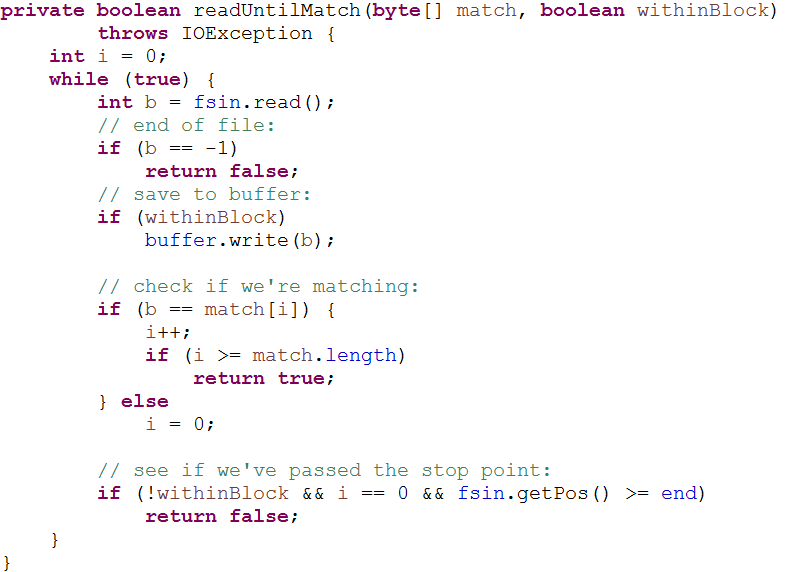


We have started the inner class XMLRecorder, which is going to read the XML file, in the constructor of XML Recorder class, we have 3 functions **getStart(), getLength() and getPath().** The getStart function returns the position of the first byte in the file to process. The getLength function, returns the number of bytes in the file to process. The getPath function, provides file containing this split's data



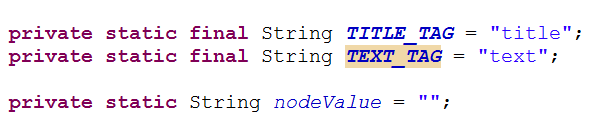
There is another function, which writes the content within the start and end tag to the buffer



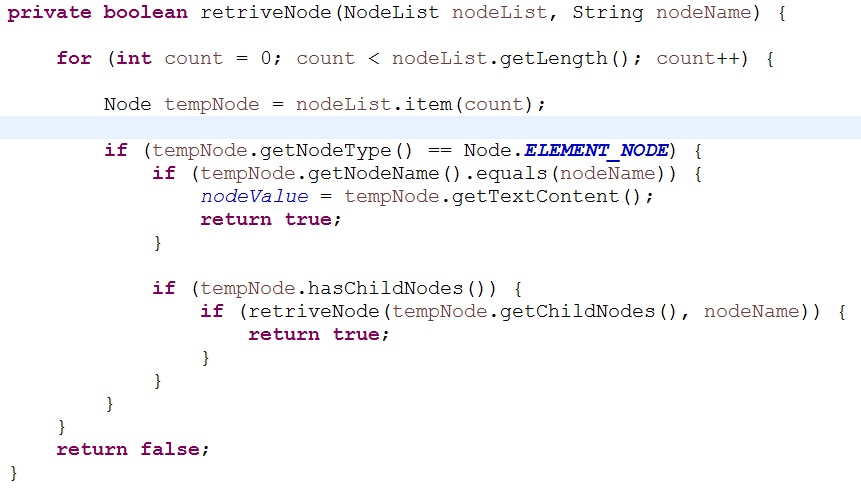


**readUntilMatch** is a utility function used by the next function, it reads the xml file until the tag is found, and once the rag is match it start writing the output to the buffer and output it.

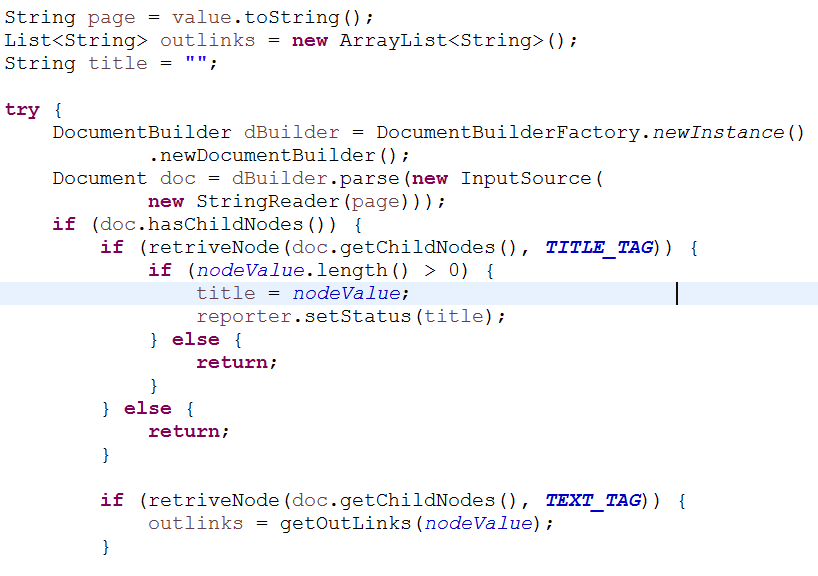
Later in **GraphBuilderMapper class**, we are setting the value of title tag and text tag, which will be helpful while retrieving the value from the document.



We have function **retrieveNode**, which is used for retrieving the node and the content within it, which we have provided. If it founds the node, it is going to set the content to the nodeValue and return true, else it will return false.



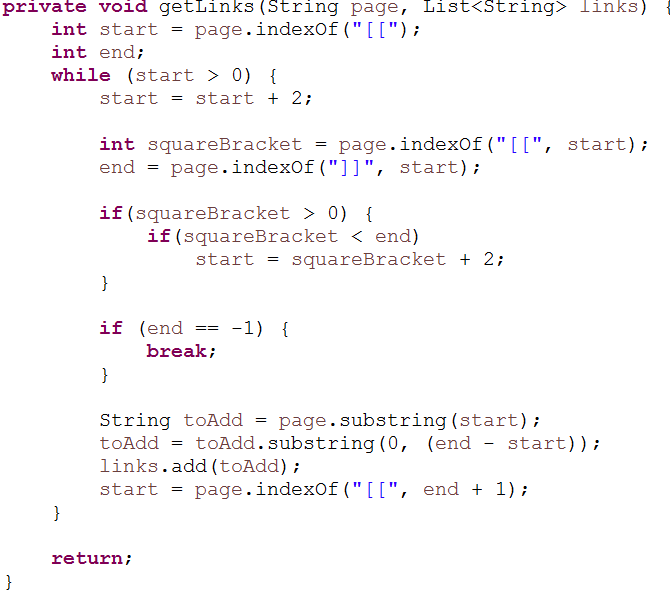
We are creating an instance of Document Builder and then creating a new document builder, which will read the text and title tags within the <page> and </page> tag, which are created by XMLInputFormat class. It will first look for the child node with the name as **“Title Tag”** which we have assigned to it in the starting, in our case it is “**title**” and it will assign the title tag value to **nodeValue**, once it retrieved the title tag, it will look for the **“Text Tag”** in our case it is “**text**” and send the retrieved text as a input to **geOutlinks function.**



In the **getOutlink** function, we are retrieving the link from the patterns which we have observed, the first pattern which we have observed is **"(== Taxonavigation ==|==Taxonavigation==)([^(==)]+)"**, the code is going to match the pattern, if pattern found, the content within the pattern is sent as input to **getLinks function** with ArrayList which will store all the outlinks from that text**,**  which will eventually created a outlinks for the URL. The second pattern observe is **"(== Name ==|==Name==)([^(==)]+)"**  and the code the same thing as above for this pattern as well.

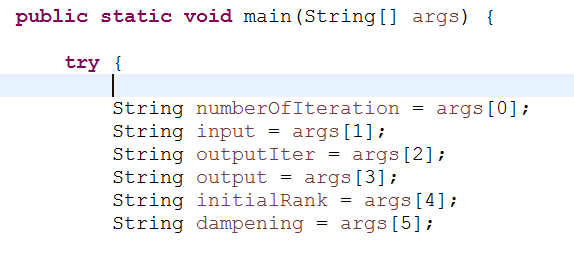


In the **getLinks** function, it take string as input and arraylist as input and later it will retrieve all the test which are within double square bracket “**[[**” and put it an array list which is later return to modify and outputted as mapper output. There are some string which has starting square bracket but not closing square bracket, to handle that condition, we are taking the start index of square bracket and taking the indext of second start square bracket and index of closing square bracket as well, if the index of second start square bracket is less than the index of end square bracket, then we change the start index to the second square bracket, and put the content within it as a outlink.

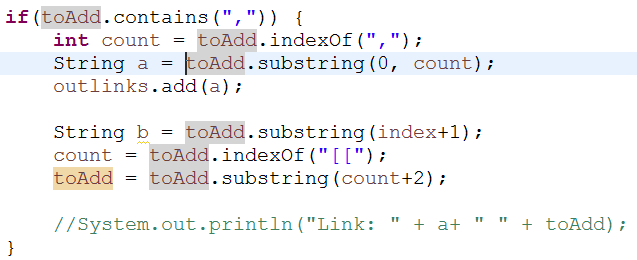


**Cleaning of title and outlinks**

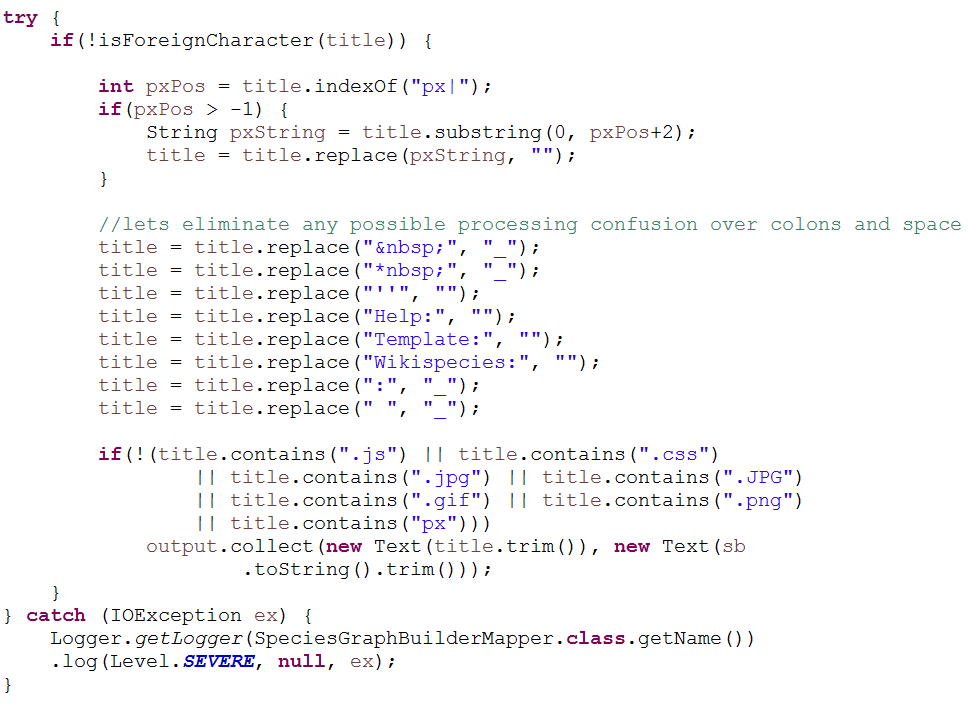
* **Dampening factor, Intial PageRank and Number of Iteration are parameterized in the code.** I have changed the d to 0.85 from 0.98 and Initial Page Rank from 1 to 0.1, as these are appropriate values.



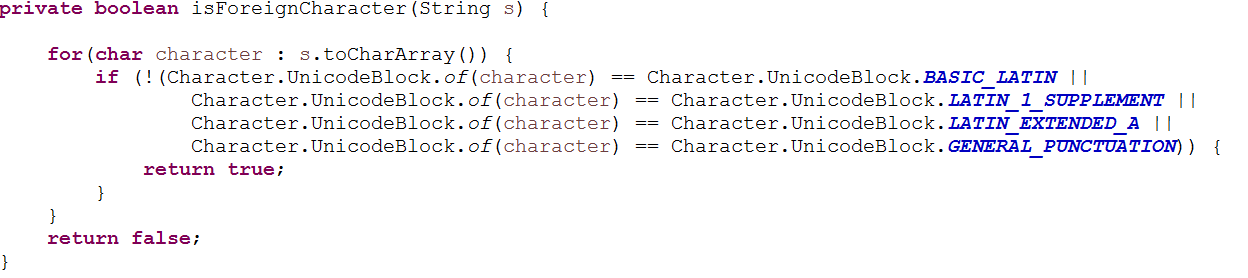
* There was few lines where there are open brackets but that are not closing, we have to take care of those cases as well. What eventually is happening is, the present code is considering one long string which has 2 outlinks as one. The code below is taking care of that thing, instead of closing square bracket there is a “,”. I am using that to retrieve the URL



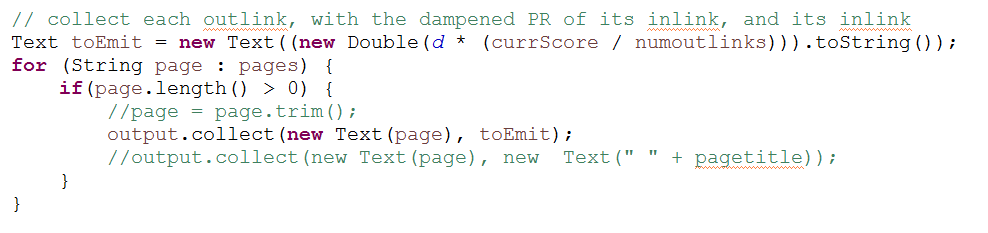
* Removed the following things **spaces, colon, &nbsp, \*nbsp,’’, Help:, Template: and Wikispecies:** from the list-of-URLs as, at many places we are splitting the data based on “ “ and “:”. So having the presence of these punctuation in the text result into the wrong computation and we were getting output as Key as Integer and value as Integer. There were presence of some different locale in the text which has “:” between the given URL, it was splitting them and they both were considered as outlink which is not the actual case, after replacing them those discrepancies were removed.
* We are also removing the title and outlinks which are representing just the **“.js”** and **“.css” i.e., javascript and stylesheet files**. Along with this we are also removing the lines which has **jpg, png, gif** since these represent the images and should not be considered as outlinks or title.



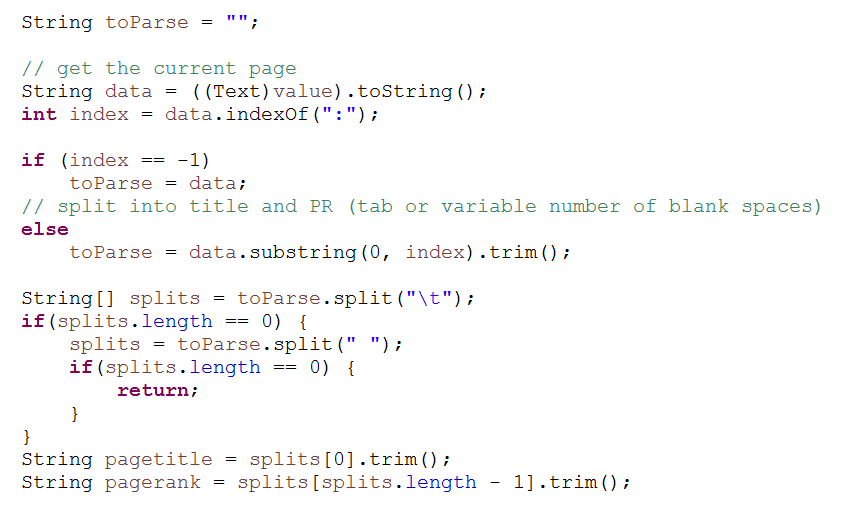
* We are also removing the title and outlinks which has characters other than **Latin**, there are few text which has text in other languages which are not useful to us.



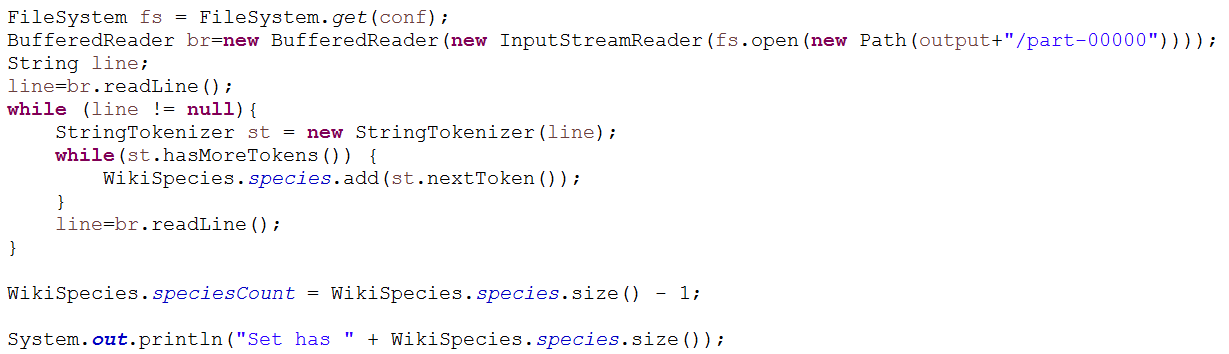
* While computing the page rank, the total number of species were not taken in account, I have computed the N and included while calculating the dampening factor as **(1-d) / N, where N is the total number of species in the document.**
* In the **SpeciesIterator Mapper,** we were outpting the inlinks as well, which is not required. We commented that line



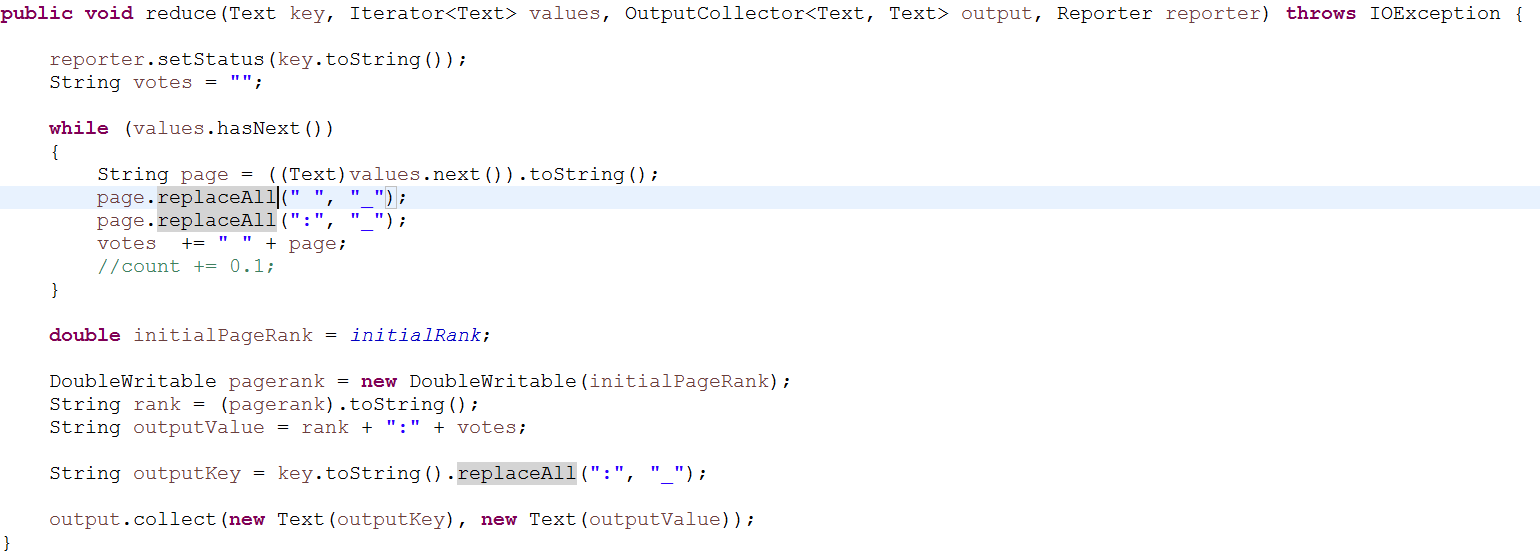
* In the **SpeciesViewer Mapper,** the code was to only read the data which has outlinks, I wrote if else loop to tackle the condition. Because the code was not there, the Mapper was not outputting the pages which didn’t have outlinks. Page with outlink will have colon and otherwise will have not have colon.



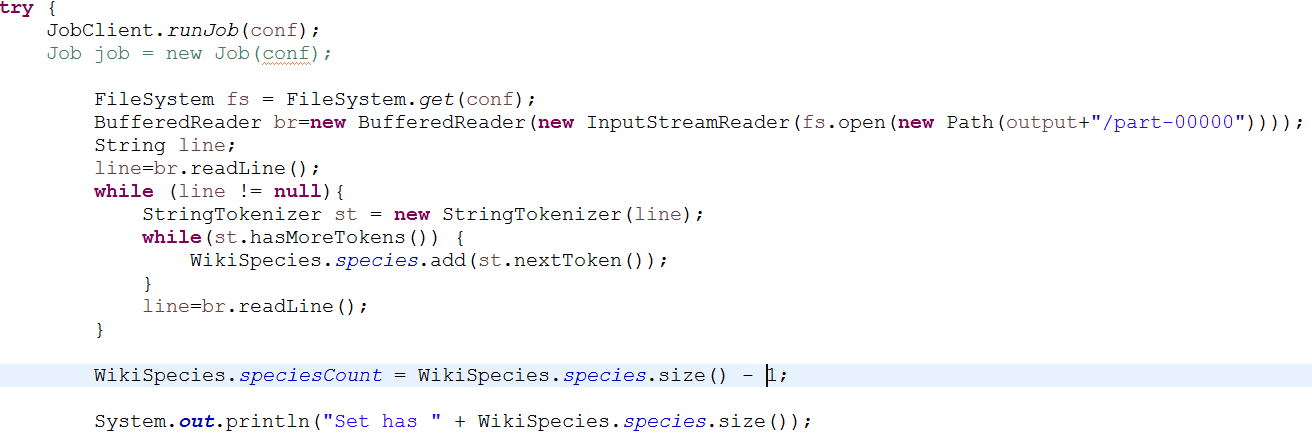
* **Reading file from the hdfs,**  I read the file from the hdfs to compute the total number of species which will be helpful in computing the Silver Surface formula.



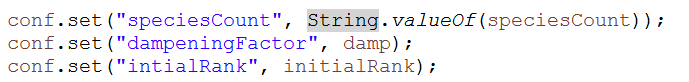
In the **GraphBuilder Reducer**, we are removing the space and colon from the outlinks and giving each URL a page rank as 0.1 (it is guessed one). The output of reducer will be **(URL, assumed page rank of the URL and the space separated outlinks)** and format will be **[URL PR: list-of-URLs]**. Assumed page rank is set by the user by passing the argument in the starting.



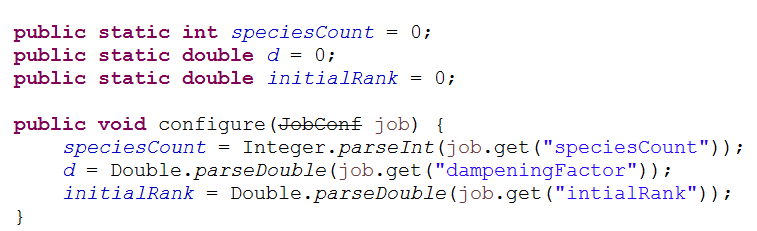
The output of the Graph Builder MapReduce will be considered as input for the species iterator MapReduce. But before that we are going to use the generated output file to count the total number of species present in the document, which we are going to use for computing Silver Surface formula. For computing total number of species, we are going to have a globally defined static HashSet, HasheSet because it does not store duplicate values, in which we are going to read the output file and store the names using StringTokenizer.



After computing the total number of species, we are going to subtract one from the size of HashSet as it will be storing 0.1: value as well, since we have used **“ “** as a token to split the line. We got total species as **376725**. The SpeciesIterator will receive the input as **URL, Page Rank and list-of-URLs** from the Graph Builder and **Total Species count, initial page rank and dampening factor** will be set in the jobconf of the Iterator Driver, which can be retrieved using Configuration method.



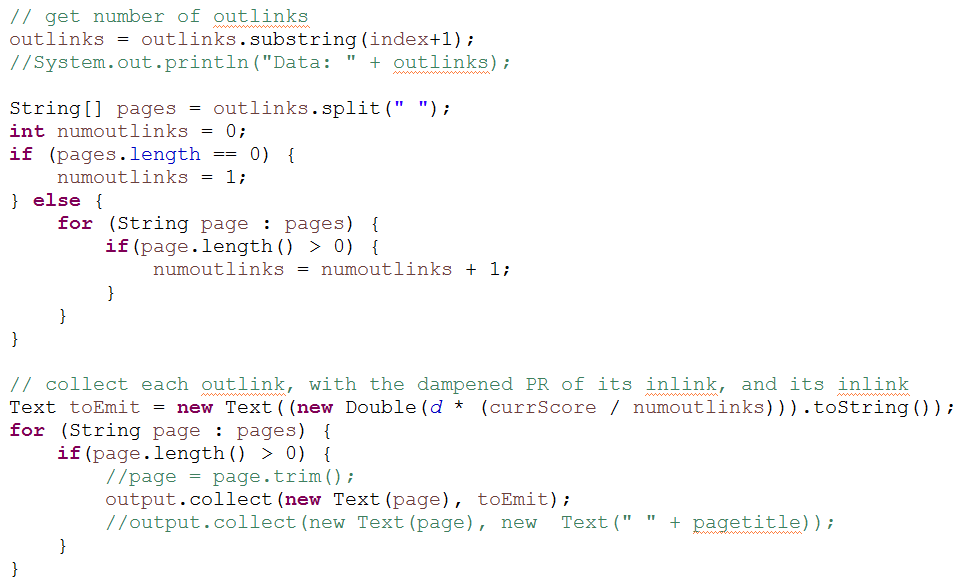
**Retrieval of the above values in Mapper**



In the **SpeciesIterator Mapper,** we are going to split the input based on colon **“:”**, where the left part is the URL and page rank and the right part will be the list of the URL’s from that web page. If there is not **“:”**in the line, we are not going to considered that line. Now we are going to split the left part of the above part based on space or tab **Page Title and Page Rank.**

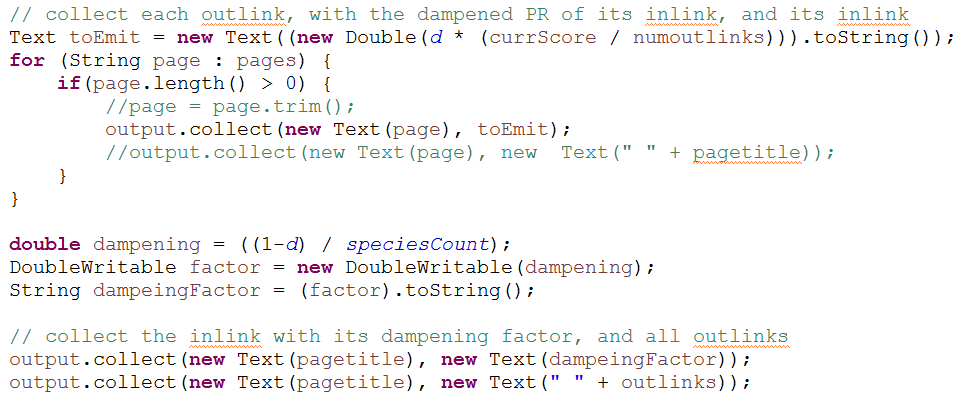


Now we are left with the string of the outlinks separated by **“ “,**  we are going to compute the number of the outlinks from the web page.

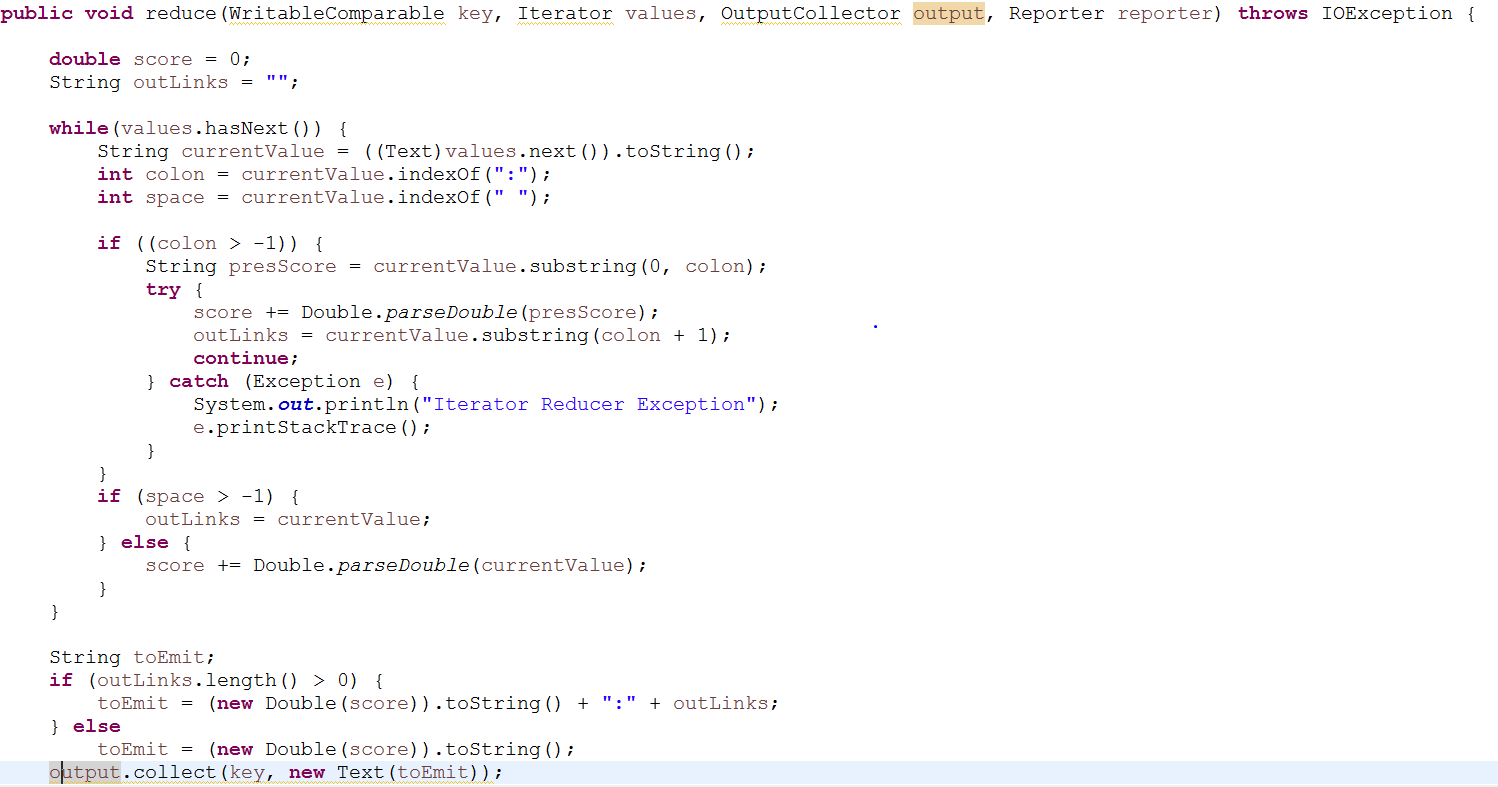


We are going to compute the **Silver surfer formula**, using the above values. We are going to assume the dampening factor and multiply it with the (Page Rank of the page / Length of the list-of-URLs) i.e., **[d \* (PR \* num\_of\_outlinks)]**.

The output of the mapper will be the URL and computed page rank, URL. We are also going to output the webpage page rank [(1 – d) / N], i.e., the probability of direct reaching the web page. These are the URLs which were input to the mapper and also output the list-of-URLs for these web pages.

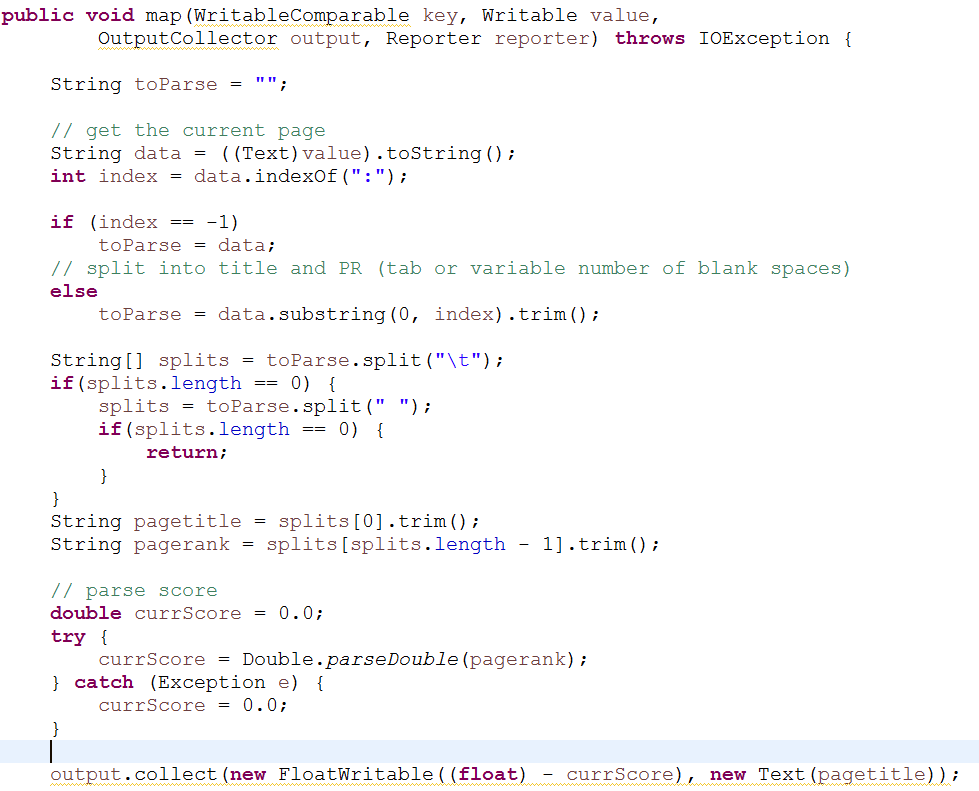


In the **SpeciesIterator Reducer**, we are splitting the data based on **“:”** and **“ “,**  to aggregate the score of the web page and separating the outlinks



The output of the reducer will be the key as URL and value as the page rank of the URL and outlinks which are colon separated. For species iterator, we are having 5 reducer tasks. This Species Iterator will be executed couple of times, till the time page rank converges completely. The converged page rank is going to act as input for Species Viewer.

In the **SpeciesViewer Mapper**, we are going to take the input from the Species Iterator, we are going to make the page rank as key and page title i.e., URL as value. We are going to make page rank as negative, so that when mapper sort the value, we can have the highest rank at the top. The output of the species viewer we conclude the page rank algorithm.



We have used if and else loop, to separate the page which have outlinks and the page which don’t have outlinks.

Since, the pages with outlinks will have colon within them and other pages won’t have colon in them.

**Conclusion**

**Euphaedra** is the species page which has the highest rank, We ran the algorithm only for 50 times in the local machine and 30 times in the fully distributed mode, due to hardware limitations, if we are going to iterate the species iterator more number of times, chances are page rank of other wikispecies pages will improve it is because convergence is completely not happened yet.

**Output**

