SWE 261P Week 3 Assignment
Group 8: Aman Bhatia, Santhiya Nagarajan, Yitian Ma

TABLESAW

Introduction

“TableSaw '' is a Java-based data frame and visualization library. The library also supports loading, transforming, filtering and summarization of data. The following are some of the features of TableSaw.

1. It allows data import from a variety of sources like RDBMS, EXCEL, CSV, JSON, HTML and text files.
2. It allows data export into CSV, HTML, JSON, and text.
3. Can combine tables, handle missing values, add or remove rows and columns.
4. Operations like sort, group and query are also supported.

Data Visualization is achieved by using a wrapper on top of the Plot.ly javascript library.

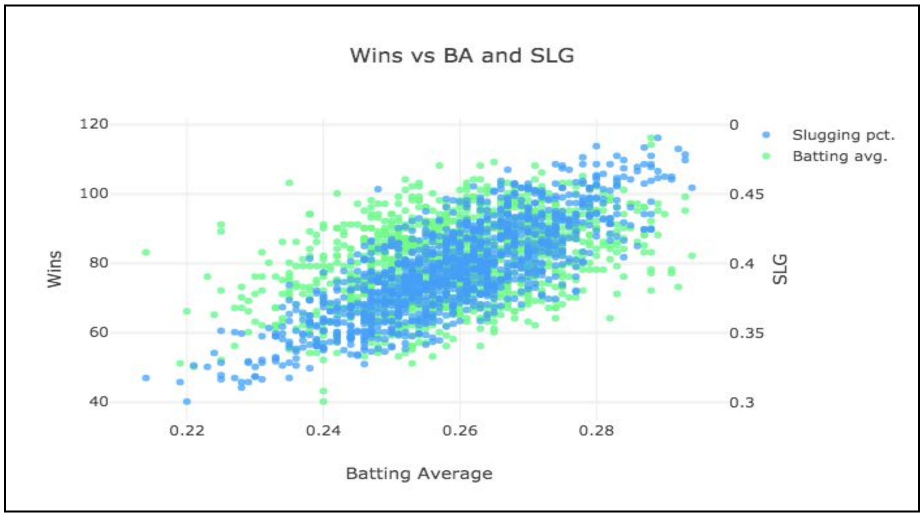
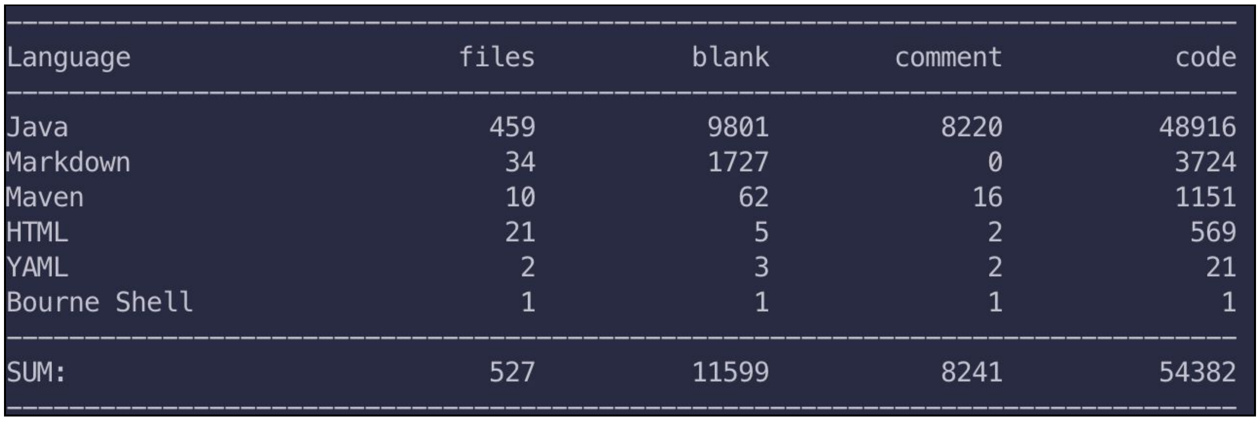


Figure (1) : Example of a data visualization plot achieved from the library

One of the main purposes of creating TableSaw came from the need of having a Java-based data science library. Java even though being a popular language wasn't designed for data analysis. Thus, TableSaw makes data analysis easy in Java.

The following are some project specifics.

1.	Build tool	Maven
2.	Languages used	Java, HTML, YAML, Markdown
3.	Lines of Code	54,382



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Java                           459           9801           8220          48916
Markdown                        34           1727              0           3724
Maven                           10             62             16           1151
HTML                            21              5              2            569
YAML                             2              3              2             21
Bourne Shell                     1              1              1              1
-------------------------------------------------------------------------------
SUM:                           527          11599           8241          54382
-------------------------------------------------------------------------------
```

Figure (2) : TableSaw LOC analyzes using Cloc

How to build and run (tests)

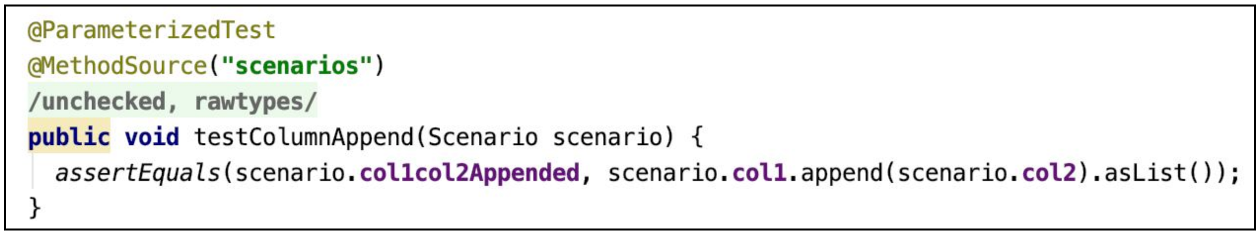
Tablesaw is a library, so there is no such entry point, but there are a bunch of tests that we can run.

1. Git clone https://github.com/jtablesaw/tablesaw.git
2. Open IntelliJ → Import Project → Select the project → Import project from the external model → Select Maven → Finish.
3. Wait a few minutes for the deployment to be complete. Click “Enable auto-import” if the prompt shows up in the bottom-right corner.
4. To run test cases, go to core → src → test → java → tech.tablesaw, right-click and run tests.
5. It is also noticed that there are some test cases in jsplot → src → test → tech.tablesaw.plotly, which are responsible for testing the output plots based on HTML and Javascript. Select the test of your interest and run it, a test output composed of HTML will show up in your default browser.

Notice that the above comprises the build in general. As this is a library, we do not have a normal main function as an entry point to run. Hence, the above addresses the build and run of the test cases.

Existing Test Cases

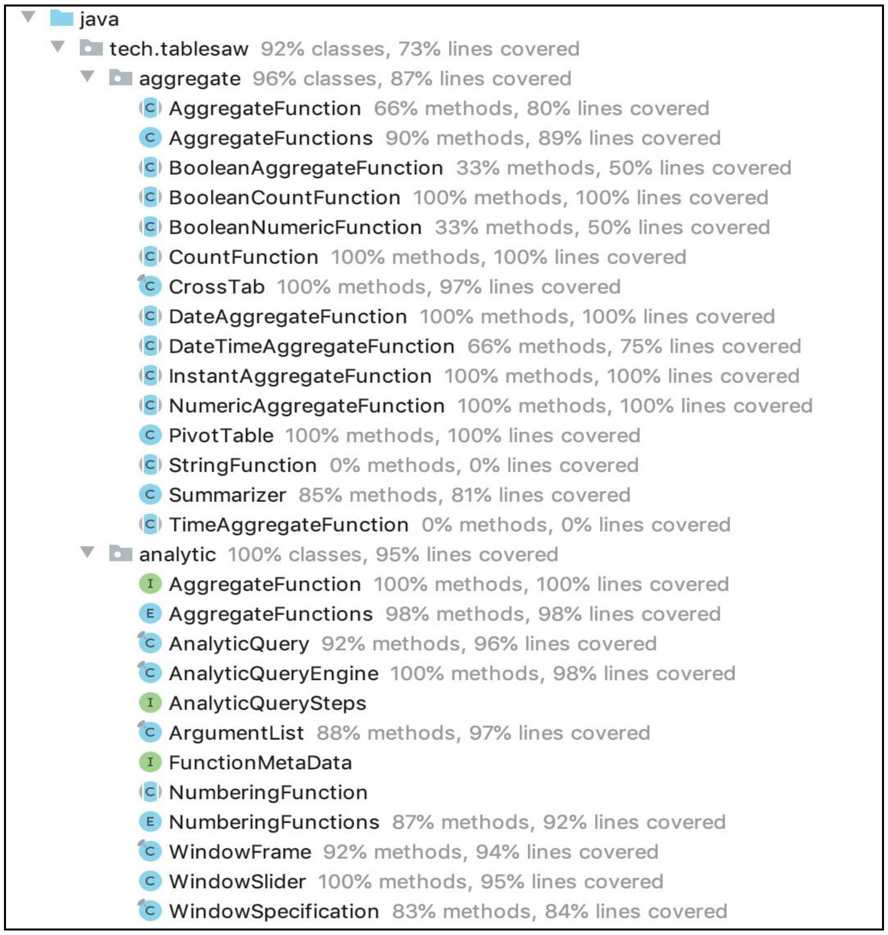
There are 101 test Java files. Each test file runs a unit test on the corresponding file. We noticed that the parameterized test was utilized only for ColumnAppend class.



```
@ParameterizedTest
@MethodSource("scenarios")
/unchecked, rawtypes/
public void testColumnAppend(Scenario scenario) {
  assertEquals(scenario.col1col2Appended, scenario.col1.append(scenario.col2).asList());
}
```

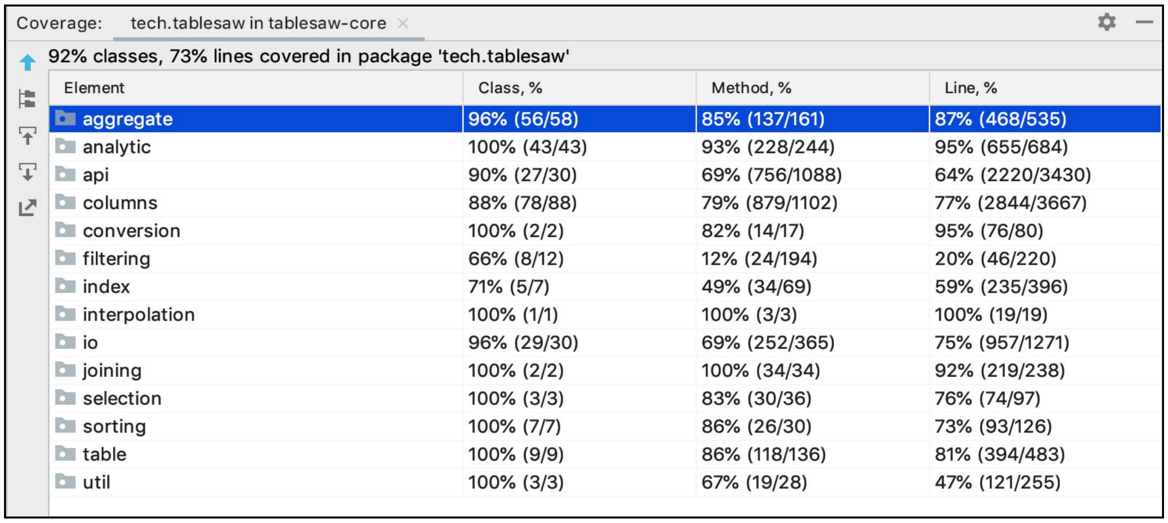
Code Snippet (1) : ColumnAppendTest.java [Lines 95 - 100]

On running the general coverage test, we noticed that 92% of classes were covered, offering us a good set of test cases, to begin with. However, on further analysis, we notice that even though these test classes cover most of the methods, with at least 2 per class, more test cases should be included to cover the entire sample space.



```
java
  tech.tablesaw 92% classes, 73% lines covered
    aggregate 96% classes, 87% lines covered
      AggregateFunction 66% methods, 80% lines covered
      AggregateFunctions 90% methods, 89% lines covered
      BooleanAggregateFunction 33% methods, 50% lines covered
      BooleanCountFunction 100% methods, 100% lines covered
      BooleanNumericFunction 33% methods, 50% lines covered
      CountFunction 100% methods, 100% lines covered
      CrossTab 100% methods, 97% lines covered
      DateAggregateFunction 100% methods, 100% lines covered
      DateTimeAggregateFunction 66% methods, 75% lines covered
      InstantAggregateFunction 100% methods, 100% lines covered
      NumericAggregateFunction 100% methods, 100% lines covered
      PivotTable 100% methods, 100% lines covered
      StringFunction 0% methods, 0% lines covered
      Summarizer 85% methods, 81% lines covered
      TimeAggregateFunction 0% methods, 0% lines covered
    analytic 100% classes, 95% lines covered
      AggregateFunction 100% methods, 100% lines covered
      AggregateFunctions 98% methods, 98% lines covered
      AnalyticQuery 92% methods, 96% lines covered
      AnalyticQueryEngine 100% methods, 98% lines covered
      AnalyticQuerySteps
      ArgumentList 88% methods, 97% lines covered
      FunctionMetaData
      NumberingFunction
      NumberingFunctions 87% methods, 92% lines covered
      WindowFrame 92% methods, 94% lines covered
      WindowSlider 100% methods, 95% lines covered
      WindowSpecification 83% methods, 84% lines covered
```

Figure (3) : Test Coverage statistics of a few classes


Coverage: tech.tablesaw in tablesaw-core

92% classes, 73% lines covered in package 'tech.tablesaw'

Element	Class, %	Method, %	Line, %
aggregate	96% (56/58)	85% (137/161)	87% (468/535)
analytic	100% (43/43)	93% (228/244)	95% (655/684)
api	90% (27/30)	69% (756/1088)	64% (2220/3430)
columns	88% (78/88)	79% (879/1102)	77% (2844/3667)
conversion	100% (2/2)	82% (14/17)	95% (76/80)
filtering	66% (8/12)	12% (24/194)	20% (46/220)
index	71% (5/7)	49% (34/69)	59% (235/396)
interpolation	100% (1/1)	100% (3/3)	100% (19/19)
io	96% (29/30)	69% (252/365)	75% (957/1271)
joining	100% (2/2)	100% (34/34)	92% (219/238)
selection	100% (3/3)	83% (30/36)	76% (74/97)
sorting	100% (7/7)	86% (26/30)	73% (93/126)
table	100% (9/9)	86% (118/136)	81% (394/483)
util	100% (3/3)	67% (19/28)	47% (121/255)

Figure (4) : Test Coverage statistics of entire package

Partition and New Test Cases

Tablesaw is a new tool for each one of us, and also has a relatively large codebase (50k LOC), it is nearly impossible for one to read each line of code and understand every functionality in detail. Hence, to test this tool, one could use systematic functional testing and partition testing. Functional testing utilizes the formal or informal program specification, which is the representation of planned program behaviors, to partition all the possible inputs. It is known that functional testing often implies systematic testing. In systematic testing, we pick possible inputs in a non-uniform fashion and focus on especially valuable inputs that tend to fail often or not at all. Compared to other testing strategies such as structural testing which focuses on the implementation details, systematic functional testing is great for detecting the logic faults because it ignores the implementation. Moving further on to partition testing where we divide the input space to smaller units which are prone to have issues, minimizes our effort needed in understanding the program. As failures in the whole program may be concentrated in a specific region, it calls for the need to use Partition testing for improving analysis.

The feature we are interested in is Interpolator (locate Interpolator.java in package tech.tablesaw.interpolation), which is to create a new column with missing cells filled based on the value of nearby cells. It has two methods, backfill() and frontfill(), both returns column type. Frontfill() fills missing values with the last non-missing value. Backfill() fills missing values with the next non-missing value.

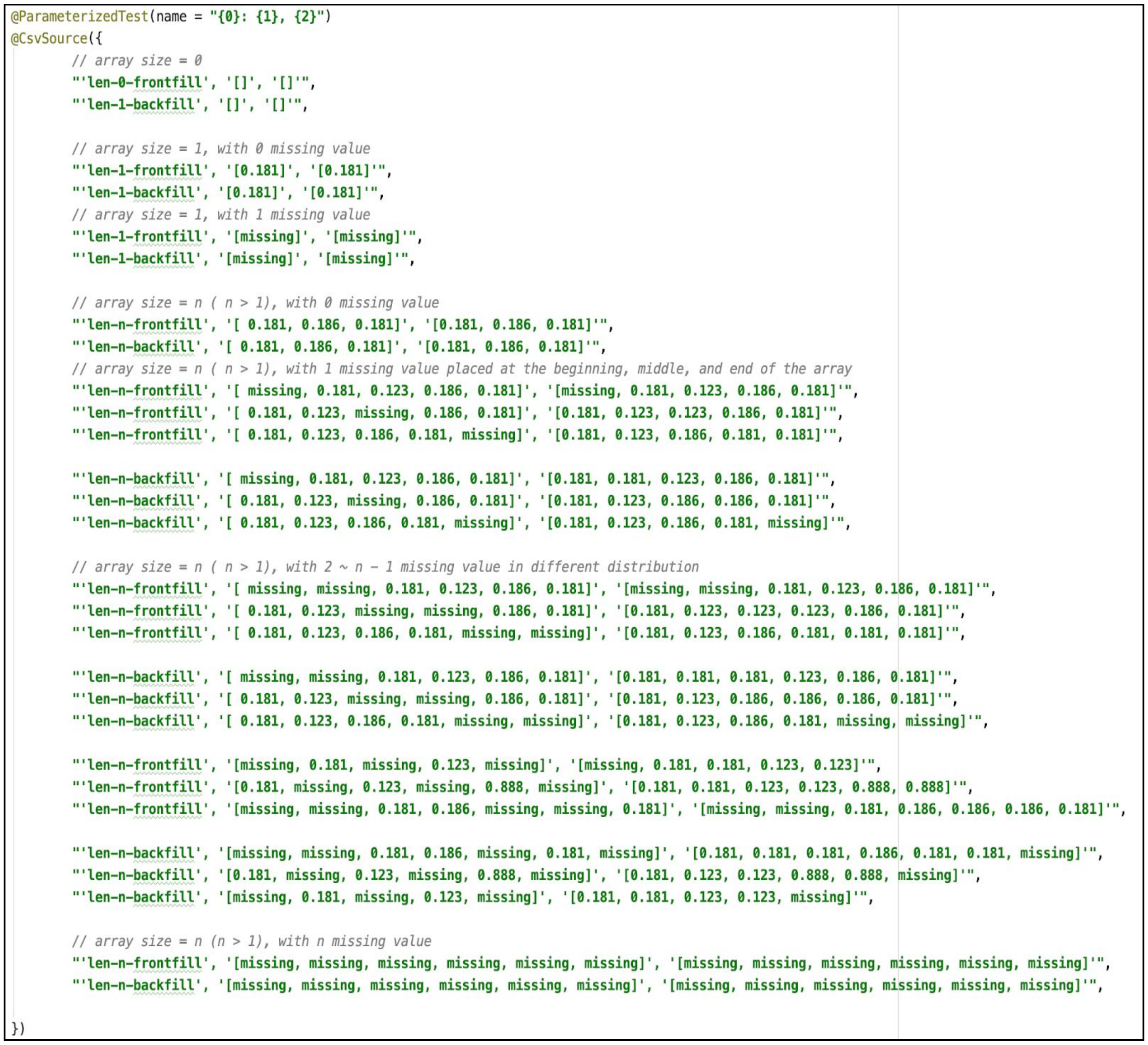
The original test (InterpolatorTest.java) provided only has two test cases to test the two methods. The input double arrays used in these test cases are random. So, we made three possible partitions of the input space:

- Length of the input double array, which is also the size of the column.
- Number of missing values in the double array.
- Distribution of missing values.

Table (1) : Appropriate boundaries for each partition

Length of the double array (N)	N = 0, N = 1, N > 1
Number of missing values (M)	M = 0, M = 1, N-1 >= M >= 2, M = N
Distribution of missing values (**m** = missing values, x = non-missing values)	- xxx**m**xxx - **m**xxxxxx - xxxxxx**m** - xx**mm**xxx - **mm**xxxx - xxxx**mm** - x**m**x**m**x - **m**x**m**x - x**m**x**m** - **m**x**m**x**m** - **mmmmm** - xxxxx

- New test cases added to the file can be found at our Github repo (Link : https://github.com/SanthiyaNagarajan/tablesaw/blob/master/core/src/test/java/tech/tablesaw/interpolation/InterpolatorTest.java)
- To run all new test cases together, Run ‘testParametrized’
- Additional cases added are attached in the code snippet below.



```java
@ParameterizedTest(name = "{0}: {1}, {2}")
@CsvSource({
        // array size = 0
        "'len-0-frontfill', '[]', '[]'",
        "'len-1-backfill', '[]', '[]'",

        // array size = 1, with 0 missing value
        "'len-1-frontfill', '[0.181]', '[0.181]'",
        "'len-1-backfill', '[0.181]', '[0.181]'",
        // array size = 1, with 1 missing value
        "'len-1-frontfill', '[missing]', '[missing]'",
        "'len-1-backfill', '[missing]', '[missing]'",

        // array size = n ( n > 1), with 0 missing value
        "'len-n-frontfill', '[ 0.181, 0.186, 0.181]', '[0.181, 0.186, 0.181]'",
        "'len-n-backfill', '[ 0.181, 0.186, 0.181]', '[0.181, 0.186, 0.181]'",
        // array size = n ( n > 1), with 1 missing value placed at the beginning, middle, and end of the array
        "'len-n-frontfill', '[ missing, 0.181, 0.123, 0.186, 0.181]', '[missing, 0.181, 0.123, 0.186, 0.181]'",
        "'len-n-frontfill', '[ 0.181, 0.123, missing, 0.186, 0.181]', '[0.181, 0.123, 0.123, 0.186, 0.181]'",
        "'len-n-frontfill', '[ 0.181, 0.123, 0.186, 0.181, missing]', '[0.181, 0.123, 0.186, 0.181, 0.181]'",

        "'len-n-backfill', '[ missing, 0.181, 0.123, 0.186, 0.181]', '[0.181, 0.181, 0.123, 0.186, 0.181]'",
        "'len-n-backfill', '[ 0.181, 0.123, missing, 0.186, 0.181]', '[0.181, 0.123, 0.186, 0.186, 0.181]'",
        "'len-n-backfill', '[ 0.181, 0.123, 0.186, 0.181, missing]', '[0.181, 0.123, 0.186, 0.181, missing]'",

        // array size = n ( n > 1), with 2 ~ n - 1 missing value in different distribution
        "'len-n-frontfill', '[ missing, missing, 0.181, 0.123, 0.186, 0.181]', '[missing, missing, 0.181, 0.123, 0.186, 0.181]'",
        "'len-n-frontfill', '[ 0.181, 0.123, missing, missing, 0.186, 0.181]', '[0.181, 0.123, 0.123, 0.123, 0.186, 0.181]'",
        "'len-n-frontfill', '[ 0.181, 0.123, 0.186, 0.181, missing, missing]', '[0.181, 0.123, 0.186, 0.181, 0.181, 0.181]'",

        "'len-n-backfill', '[ missing, missing, 0.181, 0.123, 0.186, 0.181]', '[0.181, 0.181, 0.181, 0.123, 0.186, 0.181]'",
        "'len-n-backfill', '[ 0.181, 0.123, missing, missing, 0.186, 0.181]', '[0.181, 0.123, 0.186, 0.186, 0.186, 0.181]'",
        "'len-n-backfill', '[ 0.181, 0.123, 0.186, 0.181, missing, missing]', '[0.181, 0.123, 0.186, 0.181, missing, missing]'",

        "'len-n-frontfill', '[missing, 0.181, missing, 0.123, missing]', '[missing, 0.181, 0.181, 0.123, 0.123]'",
        "'len-n-frontfill', '[0.181, missing, 0.123, missing, 0.888, missing]', '[0.181, 0.181, 0.123, 0.123, 0.888, 0.888]'",
        "'len-n-frontfill', '[missing, missing, 0.181, 0.186, missing, missing, 0.181]', '[missing, missing, 0.181, 0.186, 0.186, 0.186, 0.181]'",

        "'len-n-backfill', '[missing, missing, 0.181, 0.186, missing, 0.181, missing]', '[0.181, 0.181, 0.181, 0.186, 0.181, 0.181, missing]'",
        "'len-n-backfill', '[0.181, missing, 0.123, missing, 0.888, missing]', '[0.181, 0.123, 0.123, 0.888, 0.888, missing]'",
        "'len-n-backfill', '[missing, 0.181, missing, 0.123, missing]', '[0.181, 0.181, 0.123, 0.123, missing]'",

        // array size = n (n > 1), with n missing value
        "'len-n-frontfill', '[missing, missing, missing, missing, missing, missing]', '[missing, missing, missing, missing, missing, missing]'",
        "'len-n-backfill', '[missing, missing, missing, missing, missing, missing]', '[missing, missing, missing, missing, missing, missing]'",

})
```

Code Snippet (2) : InterpolatorTest.java [Lines 39 - 85]

Functionality Testing

In order to understand how finite functional models can be useful in testing, we must first understand what functional testing is, and what kind of tools(functional models), are available to us that can aid us in our testing practices.

Functional Testing is a type of testing technique which is used to verify the functionality of a specific component in a software system against its requirements or business specification. This type of testing technique is often put under the “Black Box Testing” since the testing doesn’t focus on the code, but rather focuses on just the functionality. It is for this reason that we have built our Finite State Machine using our software system’s(**TableSaw**) documentation.

Functional Models

In software testing, we often devise strategies or testing techniques that can aid us in finding bugs in our software system. Functional models are tools, which can help us verify our software systems logic with expected results. We explain Finite State Machines and how they can be used for testing, since we have used it to test our feature under consideration.

Finite State Machine

A finite state machine is a functional model that has a fixed number of states(**finite states**) - input and output states.
They are called so, because when we are implementing finite state machines, we only have limited memory to use. Since software systems can be complex in their design and code, finite state machines serve as an abstraction that can model simple computations in a subsystem. A common way to implement finite state machines is to use state transition diagrams. A state transition diagram has nodes(circles), each of which represents a state, and directed arrows which represent a transition from one state to another. These arrows are labeled, and the labels represent an operation, condition or event.

Finite state machines can be built independent of the source code. We can easily test a subcomponent or a feature of our software system, using Finite State Machines. FSM's clearly show the possible states and the outcomes of various transitions to these states. This means we can understand the expected behaviour of a sub component by looking at various states on given events or conditions.

Feature under consideration

We chose replaceColumn(int columnIndex, Column) as our new feature since it can be correctly modeled using an FSM.

replaceColumn works as follows:

1. The column to be replaced is deleted from the Table.
2. The new column that is supposed to replace the deleted column undergoes validation.
    a. If the new column is null then its an invalid new column
    b. If the new column has duplicate name, then its an invalid new column
    c. If the new column doesn't have the same column size as the deleted one, then its an invalid new column
    d. If all the above are not true, then our new column is valid.
3. If the new column is invalid, then an Illegal Argument Exception is thrown
4. If the new column is valid, then it is inserted into the table in place of the deleted(old) column, and the table gets updated.