2015

Climate Prediction Center

Melissa Ou
Mike Charles
Sarah Marquardt

Readme for Verification System Software (Verification Web Tool)

Last Edited 6/30/2015 by Melissa Ou.

Table of Contents

Overview

Description

This software provides a verification system that is generalized to accept any forecast and observation data within the standardized formats as specified by the software requirements (See input format information in 'Input/Output'). The verification software as a whole includes a web application for the verification web tool (referred to as the web application version) as well as the ability to run the process from a command line (referred to as the command-line version) or automated cron job to create preprocessed graphics (initiates the command-line version). The software creates charts of various verification scores based on forecast and observation data from a MySQL database. In some situations some reference information and climatologies are also utilized. This documentation is for verification system with both information about the web application and command line version.

It should be noted that this software expects forecasts and observations follow categories (ie. below, normal, above) represented by integer values (ie. forecasts have values of 0, 1, 2, 3 for EC, below, normal, and above, respectively and observations have values of 1, 2, 3 for below, normal, and above). Climatologies currently contains the % of 5- or 7-day periods in a given month, over a 30 year period (1971-2000), that a given station or gridpoint had no precipitation. The climatology data is currently only used for dry station correction/evalution purposes for extended range forecasts. This system has currently handled CONUS data only although may be expanded for other areas in the future.

There is a large amount of datasets that are used in the verification software. Typically there are various independent upstream processes that produces forecasts, observations, climatology, or other reference data in flat files. The software that would typically import these flat files to the MySQL database is included in the /scripts directory of the verification system software and is generally referred to as the 'data import' process. These scripts are expected to be ran prior to the verification software in real-time. This process was created and is currently maintained by Mike Charles (CPC) and consists of a series of Perl and C-shell scripts. The Verification System 'How To Install' document contains a section called 'Setting up the Database'.

All the current files and paths are documented in a Google spreadsheet (data inventory):

Purpose

Calculate verification scores for forecasts and produce graphics and data output of various scores with the flexibility of sub-sampling the forecasts in different ways. The VWT software provides a way to produce verification results by both a user-interactive web application and command-line version (internal). The web application enables users to dynamically produce various skill score results with many options and stratifications. The command-line version enables internal users with access to the database with data to manually and easily run the software on a command-line as well as on a cron to produce verification output. The software was developed to accept what is referred to in the software as 'settings', or variables that processing can be submitted for run-time. These settings are described further in Appendix B (settings description).

Resources

Verification tool Trac project page :
You can browse the source and also view documents, project planning documents, and tickets.
https://cpc-devtools.ncep.noaa.gov/trac/projects/Verif_System

Installation instructions for run-only (no edits to code) :
https://docs.google.com/a/noaa.gov/document/d/1mvp6Zc_RxLfLoRS9kr6qYGBmNtBcMAtE0kuZTNUIRmU/edit#heading=h.td6q1kr2zbgz

Installation instructions for editing code :
https://docs.google.com/a/noaa.gov/document/d/1vdI9g2VjPjDu0pQMIHJKNwrUdSjYVoM4g6ozW6Tqcok/edit

Data inventory
https://docs.google.com/a/noaa.gov/spreadsheet/ccc?key=0ApiFqLMmodO7dFo2bTlNZWdxZnFQa1M3bmp5Q0FyVHc#gid=0

JavaDoc API :
Web application view of documentation of all packages, classes, and methods typically used by developers. You would need a completed installed and built instance of this software on a machine, including running the build software to create the files for this (ant doc_java). To view the API, in a browser load the path of the main directory of where the verification system is located and the directory /docs/api/java. Currently, to load files (ie. to individual test accounts) on the Compute Farm:
http://www2.cpc.ncep.noaa.gov/export-3/cpccftstnfs-cp/...

Conventions Used in this Document

Typographic

Constant width	Used for literal user input, command output, and command-line options
Italic	Used for program names, file and directory names, and new terms
Constant width italic	Used for replaceable items in code and text

> command arg	Describes a command intended to be typed at the command prompt

Section Headings

Section headings should always be used to organize content in this document. The title of the document itself should use a Level 1 heading. Major topics should use a Level 2 heading, subtopics should use a Level 3 heading, and so on.

Note: each heading will show up as a new Table of Contents entry.

Code Blocks and Commands

Code blocks and commands should be constant width (courier new) and be indented.

Figure Captions

Figure captions should be size 10 italic Arial font, directly under the figure.

Features of the Verification Software

The verification software was designed to have as much flexibility as possible for scalability. The software is able to output results in many formats and is built so that other formats could be added to the code in the future. This includes allowing the verification software to be initiated by command line (thus enabling it to run on a cron) as well as in a browser (web application in form of applets). Below is an overview of features of the verification software, including the static method of initiation (command line/cron).

- Multiple initiation Methods:
    - Web application with HTML forms, including clickable maps, aggregation of some setting options, etc. (see 'Features of the verification web tool' section for specific options).
    - Command line initiation. This can be used in a cron to run the verification software automatically to update output.
- Multiple output formats:
    - The command-line version can produce results in ASCII file format and chart graphics as PNGs. Spatial maps with stations cannot be created at this time using the command-line version because it utilizes the Google Earth/Map Server which does not produce saved hard-copies of graphics. A post-processing program was created by Adam Allgood and Scott Handel at CPC that uses the ASCII files with gridded spatial data results from the verification tool as input, and produces contoured maps (PNGs).
    - The web application displays results dynamically on the web application in the form of charts or spatial maps with colored circles at station locations (Google Map). Flat file ASCII data can be retrieved from the charts by utilizing the interactive JClass chart options (right-clicking on resulting chart opens this GUI).
- Multiple output dimensions:

    - Verification statistics can aggregate forecast and observation data in time and space.
    - Results can be written to ASCII files for time and space. The reference data would reflect either dates or location names depending on the selected output dimension for the process run.
- Multiple options (settings) to calculate verification statistics for. This includes user specified time and space aggregation, different forecast sources (CPC official outlooks and model tools), score types (Heidke, Brier, RPSS, reliability), and forecast category type.
- Scalable code:
    - Code was designed and created with modular designs and objects that optimize collaborative and future enhancements and additions to the code.
- Well documented software:
    - Scripts and codes are well documented, including the usage of Javascript and Java documentation (JSdoc, Javadoc) software to view documentation easily in a browser. Perl documentation software (Perldoc) is also used for documenting the pre-processing software used to import data into the database that the verification software uses. This is discussed further in the documentation for the Perl scripts and pre-processing of data for importing data into the database for the verification software. The multiple levels of documentation and methods of developers accessing documentation significantly enhances the ability to upgrade and maintain the software without the original developers' help.
- Implemented multiple logging methods for debugging and developing:
    - Implementation of Log4J logging software, which enables multiple levels of logging. This allows a configuration file to determine the level of logging to run the process for which can significantly help debugging, while at the same time not slowing down processing for logging during operational running of the software.
    - Implementation of logging that is displayed on the web application page. The developers of this software could not find a commercial software package that allows errors and messages to be thrown to a panel displayed on the web page with the VWT, so a system was developed for this purpose. See the ' Web application logging documentation using Java and Javascript' documentation in this readme for more information. This functionality also allows a unique version of an error message to be displayed on the web page to the user that is more user-friendly and designed for the user to report to the maintainers of the software to help the maintainers debug issues. Javascript pop-up alerts of errors are also an option for use of logging in the code. These instances would often be used when a fatal problem occurred and the developer wants to make it obvious to the user that a serious error has occurred during run-time.
- Tutorial and help text implemented throughout the tool:
    - There is a tutorial tab on the top of the VWT, tool tips (implemented by JQuery) for form fields, overview, etc. to help users understand what the tool is, how to use it, and how to interpret results. Various parts of the web files are used to create these pieces of text.

Data Import Process

This software includes the scripts that are ran to import data files (flat files) of forecasts, observations, etc. into the MySQL database that the verification processing ingests. There are various process owners that maintain processes that create the upstream forecast, observation, and climatology data. These process owners are expected to produce a version of the data in such a format that the verification software requires. Information regarding the various files and owners are maintained in a Google Spreadsheet called Verif_Datasets_Inventory in the 'Verification' Google folder. More documents regarding the data import process is in the 'Data' folder in the 'Verification' folder. Documentation for the process is in the Data Handbook in the 'Data' folder.

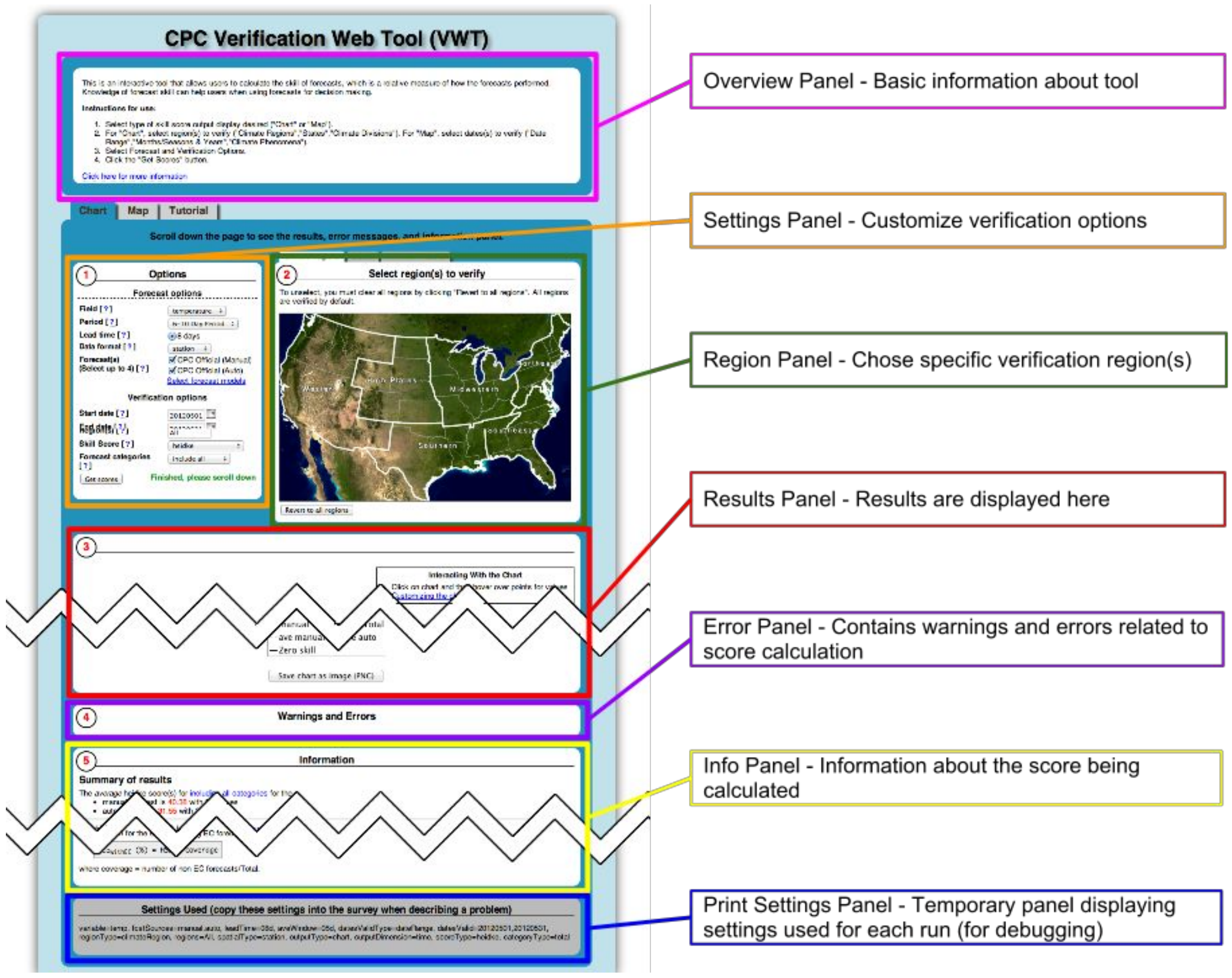
All the scripts involved in importing data into the database is in the /scripts directory of this software. The scripts are mostly written in Perl or C-shell. There are some Matlab (.m) scripts that were utilized for benchmarking as well during the testing phase.

Output overview

There are a few types of output that can be produced depending on whether the web application version or command-line run version of the tool is executed. Below are some screenshots of what output typically looks like running different versions of the VWT. Note that all dates on results are in dates valid, or the dates that the forecasts would "occur" on.

Web application version output

By selecting the "Chart" tab of the web tool, you can have verification scores displayed on a chart. Typically the chart would either be a time series or line chart for reliability, where the x-axis and y-axis have probability and frequency values, respectively. Below is a screenshot of an example chart produced on the chart tab page of the web application version of the tool. Note that you can plot multiple forecast results at once (different forecast sources but with the same settings). The dashed horizontal lines indicate the average score for each of the forecasts when applicable. It should also be noted that when "separate categories" (forecast categories) are selected, cool colors are used for the below normal category, neutral colors are used for the normal category, and warm colors are used for the above normal category (below graphic is for including all categories to make scores for each forecast source and does not depict this example)


CPC Verification Web Tool (VWT)
Chart
Map
Tutorial
Options
Forecast options
Field [?]
temperature
Period [?]
Lead time [?]
8 days
Data format [?]
station
Forecast(s) (Select up to 4) [?]
CPC Official (Manual)
CPC Official (Auto)
Select forecast models
Verification options
Start date [?]
20120501
Skill Score [?]
heidke
Forecast categories [?]
Include all
Get scores
Finished, please scroll down
Select region(s) to verify
Western
High Plains
Midwestern
Northeast
Southern
Southeast
Revert to all regions
Interacting With the Chart
Save chart as image (PNG)
Zero skill
Warnings and Errors
Information
Summary of results
Settings Used (copy these settings into the survey when describing a problem)
Overview Panel - Basic information about tool
Settings Panel - Customize verification options
Region Panel - Chose specific verification region(s)
Results Panel - Results are displayed here
Error Panel - Contains warnings and errors related to score calculation
Info Panel - Information about the score being calculated
Print Settings Panel - Temporary panel displaying settings used for each run (for debugging)

This is an interactive tool that allows users to calculate the skill of forecasts, which is a relative measure of how the forecasts performed. Knowledge of forecast skill can help users when using forecasts for decision making.

Instructions for use:

1. Select type of skill score output display desired ("Chart" or "Map").
2. For "Chart", select region(s) to verify ("Climate Regions","States","Climate Divisions"). For "Map", select date(s) to verify ("Date Range","Months/Seasons & Years","Climate Phenomena").
3. Select Forecast and Verification Options.
4. Click the "Get Scores" button.

Click here for more information

Scroll down the page to see the results, error messages, and information panel.

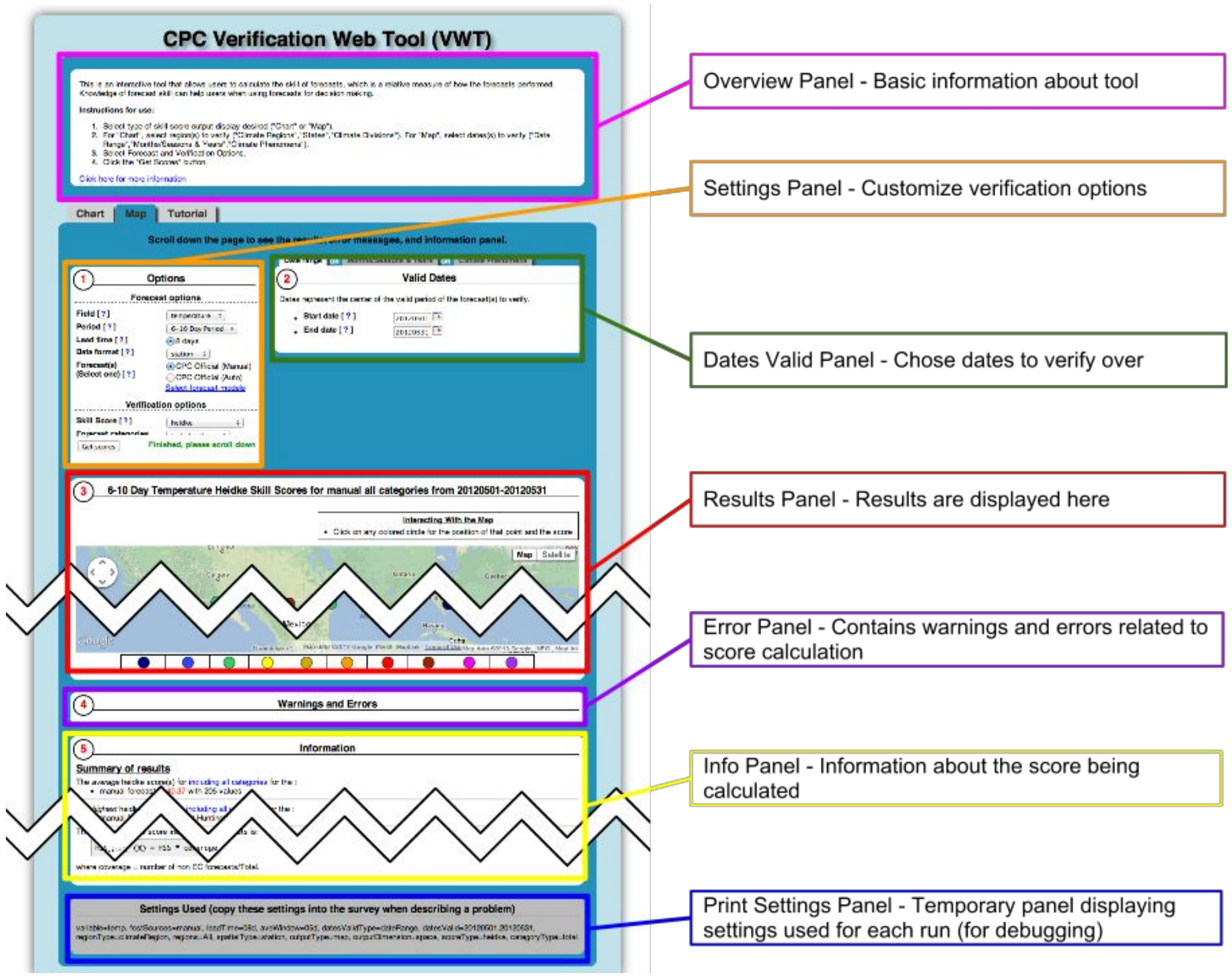
To unselect, you must clear all regions by clicking "Revert to all regions". All regions are verified by default.

Click on chart and then hover over points for values

where coverage = number of non EC forecasts/Total

```
variable=temp, fcstSources=manual,auto, leadTime=08d, aveWindow=05d, datesValidType=dateRange, datesValid=20120501,20120531,
regionType=climateRegion, regions=All, spatialType=station, outputType=chart, outputDimension=time, scoreType=heidke, categoryType=total
```

Fig.1a: Screenshot of a chart produced and displayed on the web application version of the VWT in a browser for the “chart” tab with explanations of what each part of the web page contains.


Overview Panel - Basic information about tool
Settings Panel - Customize verification options
Dates Valid Panel - Chose dates to verify over
Results Panel - Results are displayed here
Error Panel - Contains warnings and errors related to score calculation
Info Panel - Information about the score being calculated
Print Settings Panel - Temporary panel displaying settings used for each run (for debugging)
Chart
Map
Tutorial
Get scores
Map
Satellite

CPC Verification Web Tool (VWT)

This is an interactive tool that allows users to calculate the skill of forecasts, which is a relative measure of how the forecasts performed. Knowledge of forecast skill can help users when using forecasts for decision making.

Instructions for use:

1. Select type of skill score output display desired ("Chart" or "Map").
2. For "Chart", select region(s) to verify ("Climate Regions","States","Climate Divisions"). For "Map", select dates(s) to verify ("Date Range","Months/Seasons & Years","Climate Phenomena").
3. Select Forecast and Verification Options.
4. Click the "Get Scores" button.

Click here for more information

Scroll down the page to see the results, error messages, and information panel.

1 Options

Forecast options

Field [?] temperature
Period [?] 6-10 Day Period
Lead time [?] 8 days
Data format [?] station
Forecast(s) (Select one) [?] CPC Official (Manual) / CPC Official (Auto)
Select forecast models

Verification options

Skill Score [?] heidke
Forecast categories

Finished, please scroll down

2 Valid Dates

Dates represent the center of the valid period of the forecast(s) to verify.

- Start date [?] 20120501
- End date [?] 20120531

3 6-10 Day Temperature Heidke Skill Scores for manual all categories from 20120501-20120531

Interacting With the Map
- Click on any colored circle for the position of that point and the score

● ● ● ● ● ● ● ● ● ●

4 Warnings and Errors

5 Information

Summary of results

The average heidke score(s) for including all categories for the :
- manual forecast 40.37 with 205 values

where coverage = number of non EC forecasts/Total.

Settings Used (copy these settings into the survey when describing a problem)

variable=temp, fcstSources=manual, leadTime=08d, aveWindow=05d, datesValidType=dateRange, datesValid=20120501,20120531, regionType=climateRegion, regions=All, spatialType=station, outputType=map, outputDimension=space, scoreType=heidke, categoryType=total

Fig.1b: Screenshot of a map produced and displayed on the web application version of the VWT in a browser for the "map" tab with explanations of what each part of the web page contains.

Below is an example of what the reliability diagram would look like for 2 different forecast sources. The solid black diagonal line is plotted with reliability diagrams. This is a reference line that represents a "perfect" forecast.

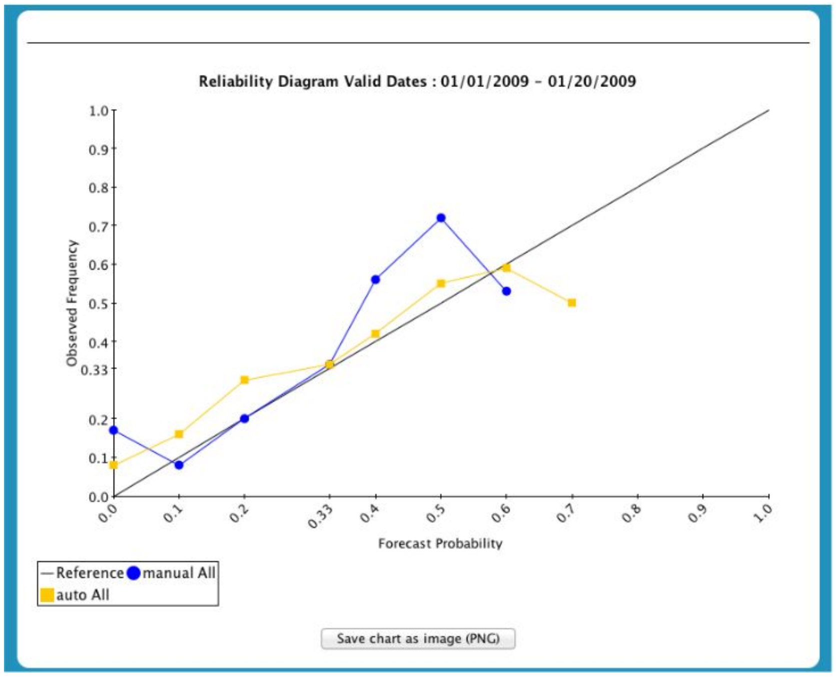
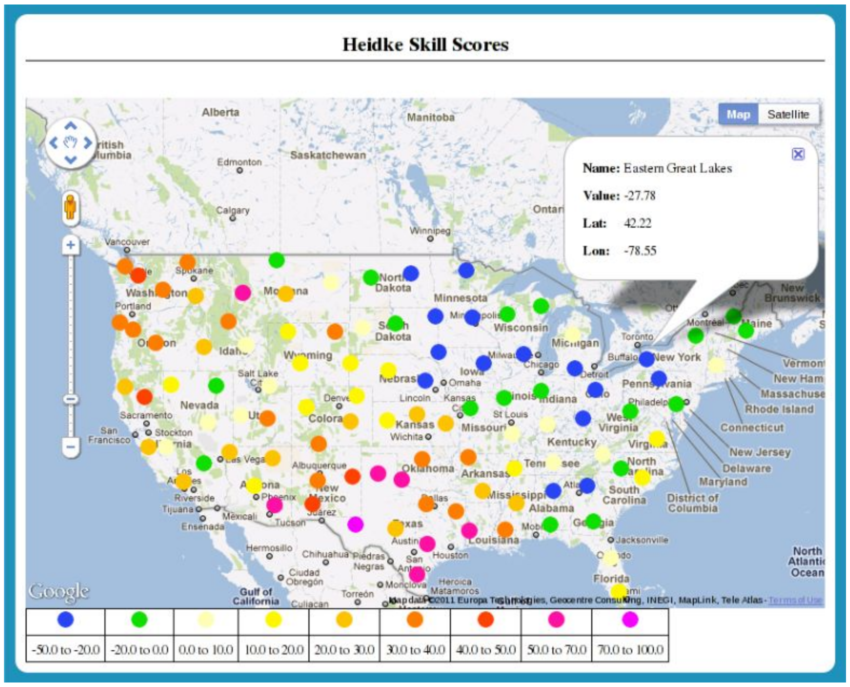


Fig. 2: Screenshot of a chart produced and displayed on the web application version of the VWT in a browser for the "chart" tab. This shows reliability skill score results for 2 different types of forecasts. Solid lines and the corresponding points are the score values. The x-axis and y-axis are in terms of probability and frequency, respectively.

By selecting the “Map” tab of the web tool, you can have verification scores displayed on a Google Map. Google Map offers a satellite overlay as an option as well as ability to zoom and pan throughout the map. Users can also click on the dots representing values and get a pop-up with the location name, latitude, longitude, and score value. Below is a sample screenshot of what a map on the web application would look like. This example shows what a pop-up would like like if a user selected a dot on the map to evaluate the information for that location.



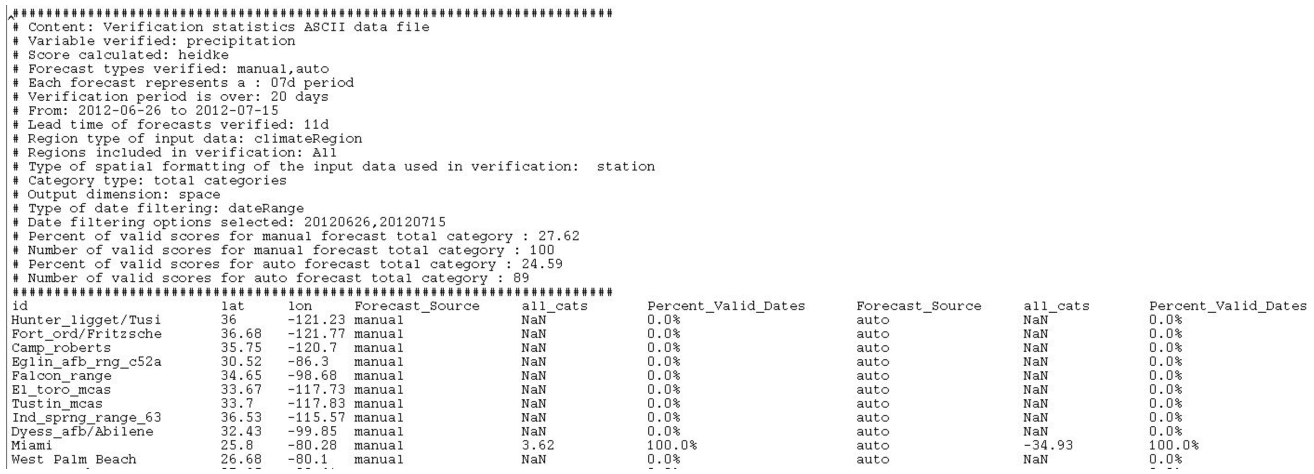
Heidke Skill Scores

Name: Eastern Great Lakes
Value: -27.78
Lat: 42.22
Lon: -78.55

🔵	🟢	●	🟡	🟡	🟠	🔴	●	🟣
-50.0 to -20.0	-20.0 to 0.0	0.0 to 10.0	10.0 to 20.0	20.0 to 30.0	30.0 to 40.0	40.0 to 50.0	50.0 to 70.0	70.0 to 100.0

Fig. 3: Screenshot of a map produced and displayed on the web application version of the VWT in a browser for the “Map” tab. This shows Heidke skill score results for one forecast. The dots represent score values (according to the legend). Values are plotted for station values.

Command-line run mode output



```
########################################################################
# Content: Verification statistics ASCII data file
# Variable verified: precipitation
# Score calculated: heidke
# Forecast types verified: manual,auto
# Each forecast represents a : 07d period
# Verification period is over: 20 days
# From: 2012-06-26 to 2012-07-15
# Lead time of forecasts verified: 11d
# Region type of input data: climateRegion
# Regions included in verification: All
# Type of spatial formatting of the input data used in verification:  station
# Category type: total categories
# Output dimension: space
# Type of date filtering: dateRange
# Date filtering options selected: 20120626,20120715
# Percent of valid scores for manual forecast total category : 27.62
# Number of valid scores for manual forecast total category : 100
# Percent of valid scores for auto forecast total category : 24.59
# Number of valid scores for auto forecast total category : 89
########################################################################
id                       lat     lon     Forecast_Source     all_cats       Percent_Valid_Dates      Forecast_Source     all_cats       Percent_Valid_Dates
Hunter_ligget/Tusi       36      -121.23 manual              NaN            0.0%                     auto                NaN            0.0%
Fort_ord/Fritzsche       36.68   -121.77 manual              NaN            0.0%                     auto                NaN            0.0%
Camp_roberts             35.75   -120.7  manual              NaN            0.0%                     auto                NaN            0.0%
Eglin_afb_rng_c52a       30.52   -86.3   manual              NaN            0.0%                     auto                NaN            0.0%
Falcon_range             34.65   -98.68  manual              NaN            0.0%                     auto                NaN            0.0%
El_toro_mcas             33.67   -117.73 manual              NaN            0.0%                     auto                NaN            0.0%
Tustin_mcas              33.7    -117.83 manual              NaN            0.0%                     auto                NaN            0.0%
Ind_sprng_range_63       36.53   -115.57 manual              NaN            0.0%                     auto                NaN            0.0%
Dyess_afb/Abilene        32.43   -99.85  manual              NaN            0.0%                     auto                NaN            0.0%
Miami                    25.8    -80.28  manual              3.62           100.0%                   auto                -34.93         100.0%
West Palm Beach          26.68   -80.1   manual              NaN            0.0%                     auto                NaN            0.0%
```

Fig. 4: Screenshot of an ASCII file produced by the command-line run version of the VWT. This is for the Heidke skill score of two types of forecasts ("manual" and "con" or "consolidation" forecasts). Scores were calculated for each of the three forecast categories (below_ave,normal, and above_ave) for both types of forecasts.

About the software

Versions

Latest available version: Version 1.0RC-1 (RC refers to "release candidate", a proposed version for release)
Current operational version: N/A

Language(s)

General description of parts of software in each language, include software versions if available:

- Web application (PHP, Javascript, HTML)
- Driver, back-end (Javac [1.6.0_20], JClass [v6.4.2])
- Output data input into chart creation methods (XML)
- Back-end database (MySQL)
- Ant build software

Installation

See the How to Install document.

Initiation/Driver Codes

This section describes codes that perform initiation and/or driver functionality of the process.

The web application of the VWT utilize two available applets which initiates the process for the chart and map web pages (tabs labeled as 'Chart' and 'Map' from the main page).

1. /source/gov/noaa/ncep/cpc/applets/SpatialApplet.java - "Map" web page.

   Outputs a spatially plotted verification score onto a GoogleMap with a dynamically updated legend below it. Score summary and errors are provided below the map. This also contains date filtering options to calculate verification score for, such as for a date range, select months all years, and select months and select years. There is also a tab for "climate phenomena" date filtering, where currently "seasonal ENSO" events are available to filter dates to verify forecasts.

2. /source/gov/noaa/ncep/cpc/applets/ChartApplet.java - "Chart" web page.

   Outputs a chart, timeseries or line chart using other reference data such as probabilities (reliability score) with a dynamically updated legend inside the chart.

3. /source/gov/noaa/ncep/cpc/applets/StaticRunDriver.java - Initiates the command-line version of the process.

   Initiates the command-line run version of the verification tool. The settings are submitted by the settings.xml file in /input instead of the settings form in the web application. There can be more than one set of settings submitted in the settings.xml file at once but can take too long to process if too many sets of settings submitted. This produces either a chart graphic or an ASCII file with data in it depending on the setting for "outputType" chosen ("chart" or "ascii"). Map graphics cannot be produced by the command-line version by this process, but Adam Allgood and Scott Handel's "Static Plots" post-process software can be used to create contoured maps from gridded ASCII output from the verification tool using GrADS. The static plots software is ran on cron currently to produce maps used displayed on the CPC verification summary web page since Google Maps (or other currently explored technology) does not have the ability to produce contour maps based on dynamically created data.

4. /source/gov/noaa/ncep/cpc/driver/VerificationDriver.java - Core back-end engine

   This is the main internal driver that performs the steps of retrieving data and calculating statistics. This class produces a loaded results object which can be accessed by either web services or the StaticRunDriver.java. The resulting loaded

Results object (Java bean) contains get methods to retrieve statistics (unformatted for the end product), score summary data, and other reference information. This class is initiated by the initiation codes listed above.

Overview of Java Software Packages

Subroutines and packages

Java packages were created based on the type of purpose that they serve. These packages provide classes and methods that perform functions that are used to produce verification score results and formatted score output. These methods and classes were coded as general as possible to be scalable and be able to be used by other Java programs. Initiation and back-end driver codes utilize these packages to perform the necessary steps for verification. One benefit of having separate packages is that it enables us to have dedicated objects that contain specific information and specific actions can be performed on them. Some of the classes and methods do not require an object of its package to utilize them. This allows other software to utilize generalized parts of the software without creating objects more specific to the verification software.
The packages included in this software, and a brief explanation of their purposes are:

- applets – Contains classes of applets that is the web application.
- data – Contains classes that is responsible for dealing with data, typically in a database, specifically MySQL. The associated data object has various get methods that enables the data that is retrieved to be accessible to other classes. This includes methods that build the appropriate Sql syntax to query the database by using the various passed settings for run-time, the actual communication with the database to retrieve data, and other necessary functions needed related to data.
- display – Contains classes that create displayed output, such as a JClass chart.
- driver – Contains classes that has typical driver responsibilities of calling methods and classes to complete run-time tasks. VerificationDriver.java is the core driver used by both the applets and the static driver, StaticRunDriver.java which is used to initiate the verification process by the command line with an input settings file in XML format.
- exception – Contains extra exceptions that can be called in addition to the default ones included with Java.
- format – Contains classes that perform a variety of format related functions. This includes:
    - Formatting arrays of calculated statistics into formats usable for output
    - Reading and writing XML (ie. reading in a settings file when using the command line initiation method of running StaticRunDriver.java, writing results into an XML format that JClass can use for charts or for GoogleEarth maps.
    - Formatting and writing statistics to an ASCII file with header information.
    - General formatting functionality (FormatLibrary.java) such as converting different dimensions of arrays to strings, rounding functions, getting an image file name.
- qc - Classes that perform quality control (QC) functionality and evaluating quality of data and scores (number of valid values).
- resources - Responsible for accessing required files.

- services - Contains classes that perform SOAP web service functionality, ie. with the Axis2 libraries.
- settings – Contains classes that handle settings which are used during run-time. Includes library functions with hash arrays that use settings values as keys for other values needed by various methods and classes in the software.
- stats – Contains classes that perform mathematical and statistical functions. Includes StatsLibrary.java , which is a library of statistical functions.
- utils - Contains functions that are somewhat miscellaneous and general, for example calculating the number of weekdays, or figuring out the climate periods spanned by the given valid date range.

The package names follow the standard Java naming convention /gov/noaa/ncep/cpc/$packageName, where $packageName is the name of the package. Classes are utilized by importing them from other classes and codes outside the package. Package information can be viewed in the API documentation.

Web Application Mode

Overview

The web application mode of the verification software enables users to dynamically select various options to aggregate the forecasts to calculate verification statistics for. This allows users to evaluate the forecasts under different situations which can help users to determine how to use real-time forecasts and how to apply them based on how they have previously performed. The options that users select may be more limited than the command line run version of the software because loading data that is too large may cause the web tool to crash or would take a very long time to process.

Features

Here is a list of features of the web application version of the verification software:

- Ability to display verification results as charts and spatial maps with dots representing score values on a Google Map
- Ability to aggregate forecasts to verify for over various settings of time and space including aggregating over seasonal ENSO events.
- Multiple types of forecasts available to verify for, including official CPC outlooks as well as forecast tools.
- Ability to save the chart as a graphic file on the user’s desktop (PNG).
- Ability to edit the look and feel and view information about the chart data and plot by clicking the right mouse button on top of the chart results. This pop-outs a GUI that enables the user to do some look and feel changes and view values.
- Ability to save data in the chart from the pop-out (right click mouse over chart results).

This can be done by going to >Data View > General. Then, at the bottom of the pop-out box is a "Save" button. The user can select the file name and location on their desktop to save the data file to. Note that this is a different format than the ASCII file that the command line run version of the tool produces.

- Warnings and errors are displayed in a panel below the results box during run-time.
- Information panel below the warnings and errors panel dynamically updates a summary of statistical results, and information about the score and results based on the updated results.
- Legends and titles on result displays update automatically according to the options selected.

Prerequisites

This software must be installed and built properly, with the proper settings in the configuration .conf files. There are .conf.example files in the repository when the software is installed to utilize as a base for the .conf which are used as run-time. Tomcat must be running on the server, the MySQL must have appropriate data, and the user must have the Java Runtime Environment (JRE) installed and enabled on their machine to load the web application.

Usage

To load the web application a user would open the location of where the application resides (with .../web/index.php in the path) in a browser. Then the user would fill out the settings (options) form on on the web page(s) as well as selecting subsets of areas on a map (optional) to verify, then submit the form. The default is set to verify the entire Continental U.S. (CONUS) unless a subset(s) is selected by the user. There are various types of geographically divided regions that can be selected (ie. climate regions, states, and climate divisions, etc.).

Process flow for the web application version

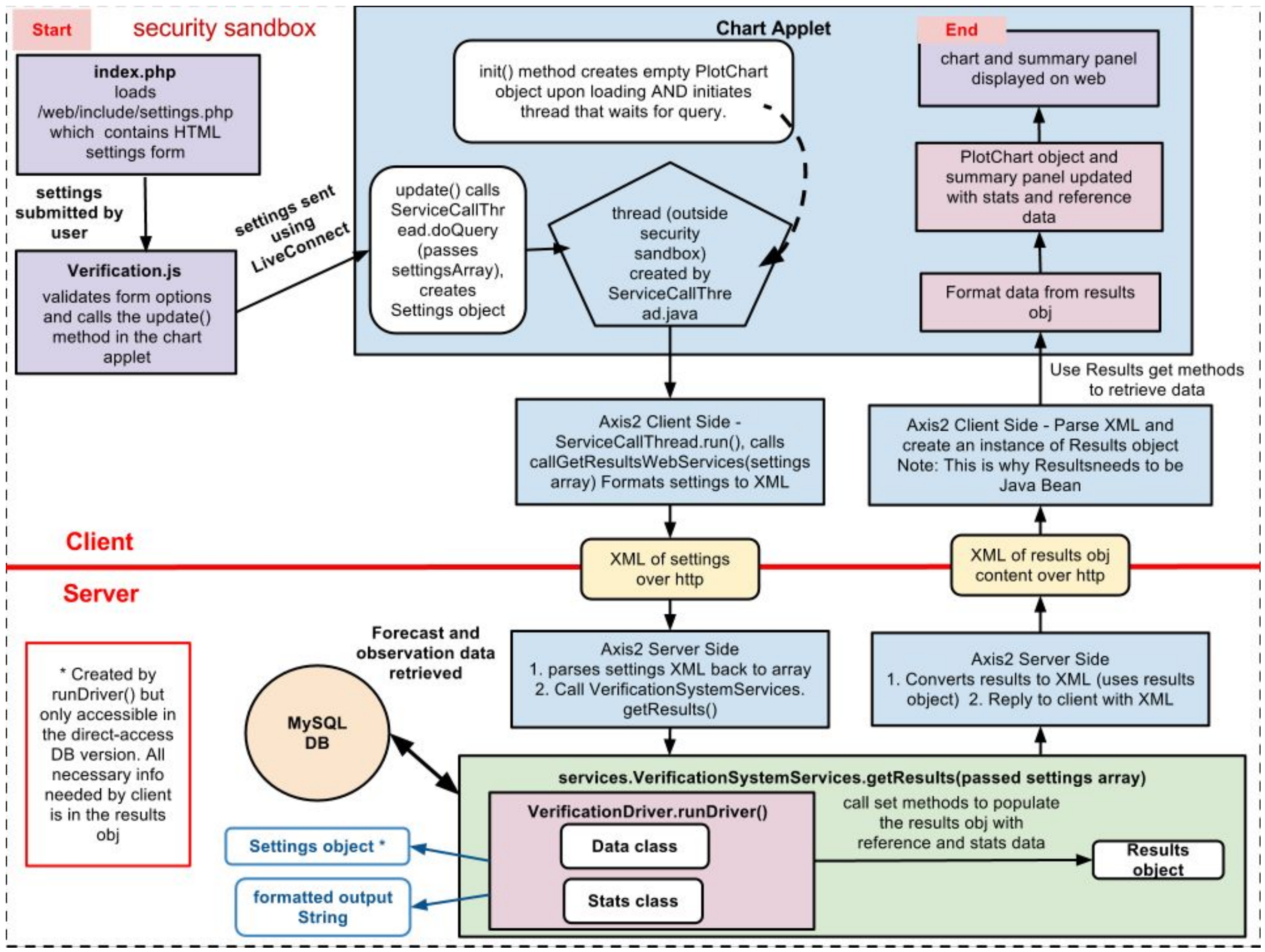

Start
security sandbox
index.php loads /web/include/settings.php which contains HTML settings form
settings submitted by user
Verification.js validates form options and calls the update() method in the chart applet
settings sent using LiveConnect
Chart Applet
init() method creates empty PlotChart object upon loading AND initiates thread that waits for query.
update() calls ServiceCallThread.doQuery (passes settingsArray), creates Settings object
thread (outside security sandbox) created by ServiceCallThread.java
End
chart and summary panel displayed on web
PlotChart object and summary panel updated with stats and reference data
Format data from results obj
Use Results get methods to retrieve data
Axis2 Client Side - ServiceCallThread.run(), calls callGetResultsWebServices(settings array) Formats settings to XML
Axis2 Client Side - Parse XML and create an instance of Results object Note: This is why Resultsneeds to be Java Bean
Client
Server
XML of settings over http
XML of results obj content over http
Forecast and observation data retrieved
Axis2 Server Side 1. parses settings XML back to array 2. Call VerificationSystemServices. getResults()
Axis2 Server Side 1. Converts results to XML (uses results object) 2. Reply to client with XML
* Created by runDriver() but only accessible in the direct-access DB version. All necessary info needed by client is in the results obj
MySQL DB
services.VerificationSystemServices.getResults(passed settings array)
VerificationDriver.runDriver()
Data class
Stats class
call set methods to populate the results obj with reference and stats data
Results object
Settings object *
formatted output String

Fig. 5 Process-flow diagram of the web application that creates charts.

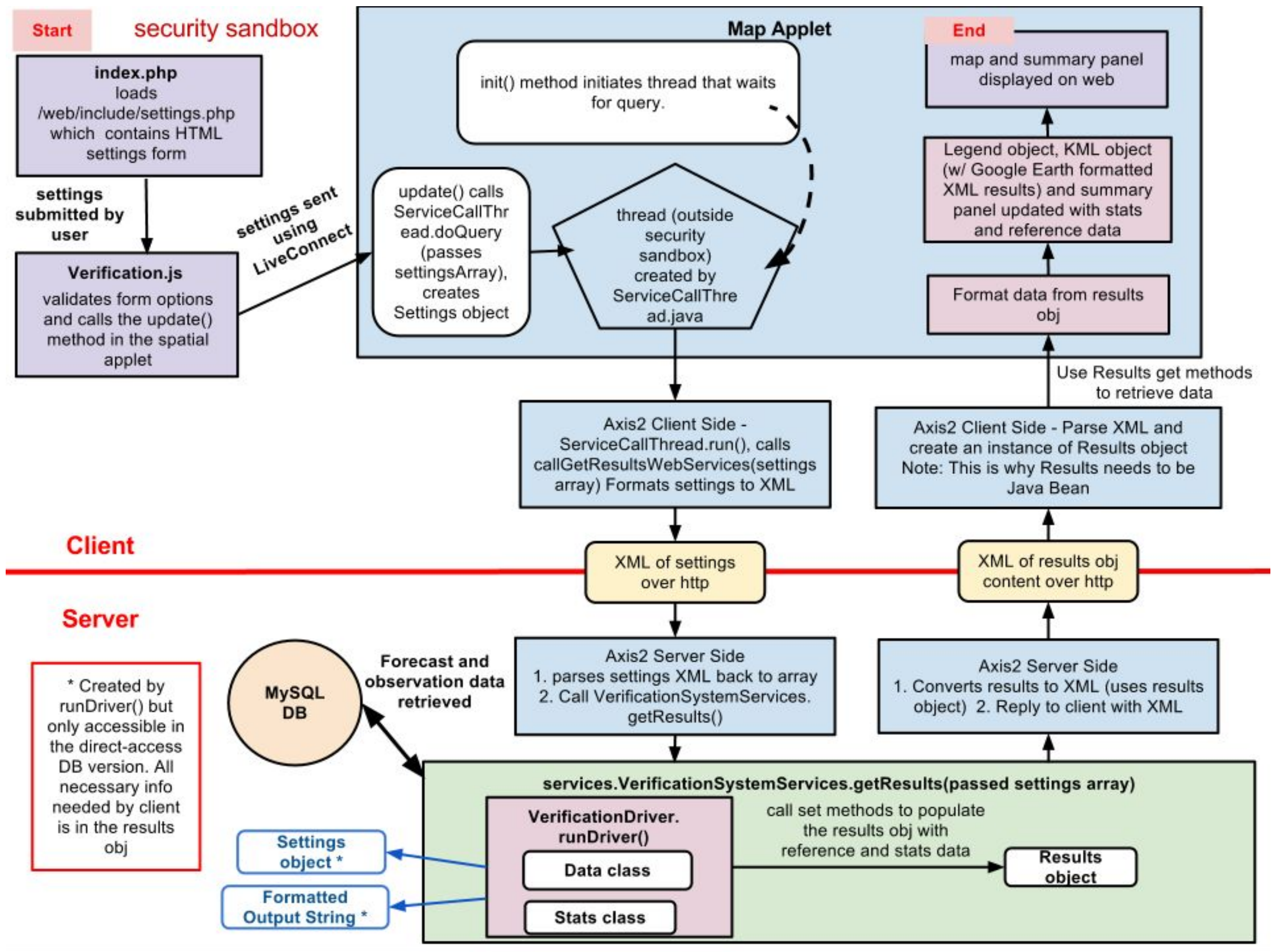

Start
security sandbox
index.php loads /web/include/settings.php which contains HTML settings form
settings submitted by user
Verification.js validates form options and calls the update() method in the spatial applet
settings sent using LiveConnect
Map Applet
init() method initiates thread that waits for query.
update() calls ServiceCallThread.doQuery (passes settingsArray), creates Settings object
thread (outside security sandbox) created by ServiceCallThread.java
End
map and summary panel displayed on web
Legend object, KML object (w/ Google Earth formatted XML results) and summary panel updated with stats and reference data
Format data from results obj
Use Results get methods to retrieve data
Axis2 Client Side - ServiceCallThread.run(), calls callGetResultsWebServices(settings array) Formats settings to XML
Axis2 Client Side - Parse XML and create an instance of Results object Note: This is why Results needs to be Java Bean
Client
XML of settings over http
XML of results obj content over http
Server
Axis2 Server Side 1. parses settings XML back to array 2. Call VerificationSystemServices. getResults()
Axis2 Server Side 1. Converts results to XML (uses results object) 2. Reply to client with XML
MySQL DB
Forecast and observation data retrieved
* Created by runDriver() but only accessible in the direct-access DB version. All necessary info needed by client is in the results obj
services.VerificationSystemServices.getResults(passed settings array)
VerificationDriver. runDriver()
Data class
Stats class
call set methods to populate the results obj with reference and stats data
Results object
Settings object *
Formatted Output String *

Fig. 6 Process-flow diagram of the web application that creates maps (by Google Maps).

The high-level steps of the web application are as follows:

The process flow steps are listed below, including slightly different steps for the chart and map applets.

1. Web application in browser – A user loads a PHP page (index.php) loads either map.php or chart.php depending on whether the user selects a chart or map via the tab on the web page. These pages contain a settings form that enable users to make selections of settings to use for running the verification software. The chart.php and map.php scripts loads the appropriate applet code base location (either the ChartApplet or SpatialApplet class), the name of the Jar file with the software (verif_client.jar). The pages contain the appropriate panels for the settings form, displayed results, and error and summary information. This index.php page calls many pieces of JavaScript and other PHP files for the map and chart display functions and handling the form information, including passing form information to the back-end Java.
2. Javascript/JQuery classes and methods – The classes and method in Javascript/JQuery format, validate, and submit the final settings required by the

back-end Java code as needed by the applets and the driver code. This could include submitting additional settings not passed in step 1 (HTML/PHP side) that the user viewing the web application does not see (ie. variables for output dimension, etc.). The verification.js Javascript has an update method that passes the settings to the Java applet when the settings are submitted. LiveConnect technology is used, which is a feature of web browsers that allows Java and JavaScript software to intercommunicate within a Web page, thus enabling the applet to be invoked by the web page.

1. Applet code – These java codes (ChartApplet.java and MapApplet.java, used for the chart and map displays, respectively) can be thought of as the high-level driver of the process. This serves to instantiate threads and call appropriate methods that create the results and display them on the web page.
2. Driver – This Java code (VerificationDriver.java) performs the main functions to retrieve the results, which is then passed back to the applets which format and display the results. This driver is utilized by both the web and command-line versions, with both the direct-access or web service methods of accessing the database. This calls appropriate methods to retrieve data from the database, calculate the statistics, and calls set methods in the Results object to set data to be accessible by the results object. The Results object has conventions of a Java Bean, which has many get and set methods. The purpose for this is to enable both web services and direct-access database methods to use this driver to perform necessary steps to get results.
3. Back-end Java packages – These are groups of Java classes and methods that perform multiple functions that the driver and applet codes access. Java objects and packages were designed in this software to make the code easier to understand, maintain, debug, and upgrade. Each package (directory of classes) performs specific roles in the verification process. See ‘Overview of Java software packages’ and the Appendices with individual package documentation for more information.

Applet process flow

1. The init() method of the applet is loaded by default upon loading the webpage prior to submitting any form options to actually produce a chart. This performs the following (in order):
   - Configures and initializes the log4j logger
   - A new thread is created by calling /services/ServiceCallThread.start(). JavaScript code is treated like unsigned code. When a signed applet is accessed from JavaScript code in an HTML page, the applet is executed within the security sandbox. This implies that the signed applet essentially behaves likes an unsigned applet. The ServiceCallThread is started here because the thread will then be able to do queries outside the security sandbox in this instantiated thread when the applet is signed. This thread will wait until it is notified to start processing a query, which occurs when Javascript initiate the update() applet method and just hands the thread the settings array. See process flow diagrams (Fig 6 and 7). This thread will wait until it is notified to start processing a query.

     It should be noted that there is one thread instantiated per applet per instance per user on the client’s machine. The thread is typically cleaned up by Java by default when the application completes running. There is a serviceClient.cleanupTransport() that gets called at the end of processing in ServiceCallThread.callGetResultsWebService().

For the ChartApplet.java only :

- Creates instance of the html file used for displaying initial text
- Sets the Swing components (contentPane) that the plot would be displayed on
- Creates a PlotChart (JClass chart) object
- Executes makeChart() in PlotChart.class which sets default chart settings including the legend without displaying anything except for text to “Submit score options”, which is set by String labelTitle in PlotChart.class.
- Adds the plot chart object (prior to chart containing results, contains object with default settings)

2. After the user selects options and submits the HTML form, the update() method is executed with the passed settings parameters (see Appendix B for more information about settings variables). It performs the following:
    - Empty the messaging panel for a new run (#errorPanelText is the CSS selector of the page element that contains potential error messaging.)
    - Calls and passes the settings array to class Results method doQuery in ServiceCallThread.java. Method doQuery() creates a query object using the settings passed, hands the query object to the thread, and notifies the thread that there is a query to be processed. The wait() function waits until there is a result. Once results are done, the results are returned to the thread.
    - Once the update method notifies the thread that there is a query to process, the run() method in ServiceCallThread.java runs, which
        - clears any previous queries,
        - calls callGetResultsWebService(settings) (in the same method as run())
        - utilizes/invokes the Axis2 web services to format the settings as XML which is passed over http to the server side internals. The class is specified for Axis2 to use in the ‘new QName’ specifications line in callGetResultsWebService(). This utilizes the build.xml specification that calls VerificationSystemServices.getResults().
    - VerificationSystemServices.getResults(settings) now performs much of the heavy lifting in terms of acting like a driver. It will call necessary parts that eventually returns a results object with the loaded necessary reference and statistics data. Note that the forecast, observation, and climatology data is not retained in the results object to minimize the size of the object that needs to be passed between the client and server. This method performs the following functions:
        - Database preferences are loaded from a configuration file (verif.conf)
        - Database connection is made
        - Data object is created, database connection set
        - Verification driver object is created, database connection is set
        - The runDriver(settings) method is called and passed the settings. Details of steps of what happens in the runDriver() step is discussed further in the Driver Process Flow section.
        - Results object is retrieved from the verification driver object
        - DB connection is closed if the thread is at the end of its life cycle.

- The run() method now has the results and sets the results in the query object. Then the thread is woken up that was waiting on this specific result. Processing continues in the update() method.
- In update() the statistics and reference data are retrieved from the results object and formatted properly. Currently available formats are an XML string for either the JClass plotChart object to display in the case of the chart, or to pass to the Google Maps server in the case of the map.
- For the chart (ChartApplet.java) :
  - The Plot Chart object set methods are accessed to set results and settings data. This updates the content that the chart object has.
  - InvokeAndWait (Intrinsic applet function) method ConfigureChart() in ChartApplet.java is passed the plot chart object and XML string with necessary data to update the chart with data. All necessary aspects of the JClass plot chart and summary information is updated on the web page.

  For the map (SpatialApplet.java) :
  - Legend object is created utilizing colored circle graphic files and threshold arrays. This legend object is formatted in HTML which is then passed to the Javascript to display on the web page.
  - Create an object (called kmlObj because it is considered KML data for Google maps server) from the formatted XML string with results data.
  - A Javascript object, called 'googleEarthClass' is created that allows the KML object with the results to be passed to the web and updating the Google map graphic. The applicable lines in the code looks like :

```
        googleEarthClass =(JSObject)
window.getMember("GoogleEarth");
        googleEarthClass.call("updateGoogleEarth",kmlObj);
```

    This data is passed to the Google Earth/map external server to update the map of results.

- For both the chart and map applets, update() gets the summary information by passing the results and settings objects to necessary methods. The score summary array is retrieved from the stats object, then formatted into HTML for display. There is no score summary for the reliability score. In the case of the map (SpatialApplet.java), there is no reliability score anyways. However, in the case of the chart applet, there is logic in ChartApplet.java for getting the summary HTML text for reliability, which is different than the rest of the score text. The rest of the scores are formatted similarly. Note: At this time the command-line version of the system does not utilize the score summary feature.
- For both the chart and map, there are HTML files which contain score information (about the score, not the values) which are read in and included in the information display. These files are located in the /input directory.
- Create an object (called htmlObj because it is data formatted in HTML) containing summary and information text and passed back to the Javascript Object that contains the appropriate window (panel) to display text on.
- The htmlObj with the information to display is passed to a Javascript object for

the web page to display. The line in the code looks like :
JSObject verificationClass = (JSObject) window.getMember("Verification");

About the Back-end Web Services

There is a layer of software that interacts between the web application of the verification tool and the database. The server is required to run Tomcat for SOAP web services to be ran. The Java Servlet utilizes Axis2 software (3rd party software) to run SOAP web services. There is a .jar file that contains software that resides on the client's machine and a .war file that resides on the server. The .jar is automatically downloaded to the user's machine upon loading the applet in a browser. There are more details in the 'Web Services' below.

Damian Hammond from the University of Arizona largely developed a solution for the requirement from NCO (NCEP IT) that the database access from users must be more secure and cannot allow direct access to the database. Therefore Damian developed a web services solution for us utilizing Tomcat, Axis2 software, Java Beans, and usage of threads. Below is Damian's description of the problem and solution used for the web version of the tool to give some background on the process:

Background Problem: The way in which the ChartApplet and SpatialApplet make calls to the Verification Tool web services layer is a little complicated. This is due to the fact that the trigger for the applets to update their data sets is provided through javascript calls to the applets update() method after the user has entered the desired criteria. In general when javascript makes a call to any method of an applet the execution path is considered untrusted and is executed inside a security sandbox. This means that any method that javascript calls is bound by the same security restrictions as an unsigned applet, even if the applet has been signed by a security certificate and marked as trusted by the user. The Axis2 libraries we use accesses system resources and cannot be executed inside the security sandbox. This means that update() cannot make calls to the Verification Tool web services directly.

Solution: To allow the applets to call the Verification Tool web services we split the actual work of calling the services and the request to call the services into two separate pieces. The ServiceCallThread class does the actual call to the web services. It is a thread that is waiting for requests to call the web services. This thread is instantiated and started in the init() method of the applet. This guarantees that the thread itself will run outside the security sandbox if the applet is signed. The ServiceCallThread provides a method doQuery() which places a request on a queue and blocks until the thread picks up the request, processes it, and returns the results of the request. Because doQuery() does not access any system resources, it just places the request in a place that the thread can pick it up, doQuery() can be executed inside a security sandbox while the thread runs outside the security sandbox.

Input

Localized data sources

1. /input/initTextDisplay.html
    a. Purpose : Text to display on the bottom panel when the page is first opened. This is an optionally used file. Currently the logic to read and use this file is enabled in the VWTApplet.class but the logic to display it is commented out. To display this text uncomment the line in the applet :
        i. textPane.setText("<html><span style=\"font-size: 20pt\">" + htmlString + "</html>");
    b. Format type (.png, .dat(Ascii), etc.): HTML text
    c. Format information/standards (Formatting of file, naming convention, headers, detailed information about format of data): none
2. /input/ERFText.html
    a. Purpose : Text to display on the bottom panel when the score chart is displayed. This is an explanation of the chart and how to read it. This is generalized, and does not involve information about specific results of the score processing.
    b. Format type (.png, .dat(Ascii), etc.): HTML text
    c. Format information/standards (Formatting of file, naming convention, headers, detailed information about format of data): none
3. input/logConfigApplet.txt
    a. Purpose : Configuration file for logging log4j processing for the web application version. This configuration is read into the process once, at the beginning of the processing. See more information about this file in the logging section of documentation in this file.
    b. Format type : ASCII text file.
    c. Format information / standards : Syntax follows log4j convention.

4. input/verif_client.conf and input/verif_server.conf
   Create these files in the /input directory by copying the verif_client.conf.example and verif_client.conf.example files and editing as necessary following the 'How to Install' document directions. Contains necessary settings such as for the database and QC thresholds, with each of them containing settings required by the client and server sides separately. These are accessed when using web services in either the web application or web services version of command-line running.

5. html files
   There are many HTML files, so they are not listed here. There are HTML files containing text for various parts of the web display, including score information for the bottom web application panel, processing text, etc.

Operational input (centralized data sources)

File path/name: MySQL databases and tables. See the data description section in Appendix C. Inventory available at
https://docs.google.com/a/noaa.gov/spreadsheet/ccc?key=0ApiFqLMmodO7dFo2bTlN

ZWdxZnFQa1M3bmp5Q0FyVHc#gid=0
Format type: MySQL database
Purpose: Provides the forecast, observation, climatology, and look up reference information.

Output

1. Client-side logs - Printed to the client's Java console if enabled and displayed. By default, the client does not have this open. It is typically used for developers trying to debug.
2. Server-side logs - Contains output from any server-side Java code executed during a command-line web-services run and when run as a web tool. See the "How to View Logs" section for more information.
3. Result graphics are displayed in the web browser on the applet. The JClass interactive GUI accessible by the chart in the applet enables users to download the data in ASCII (In JClass default format) as well as the JClass settings, although trying to use the settings to create a repeated image has not worked properly yet.

Saving charts and data

The technology behind the charts are JClass 3rd party software. This software has various functions that can be accessed from the chart GUI. Below are some save options and how to use them.

Web application save options:

1. Saving a graphic file of the chart – Once a chart is plotted on the web application, a button will appear at the bottom labelled "Save chart as graphic". This enables a user to save the displayed chart directly to the client's machine. Currently the type of graphic file is set to save as a Portable Network Graphics file (PNG).
2. Saving all the JClass Chart XML/HTML **Bug found, do not use yet** – The markup representing the JCLass chart, including the chart settings and data can be saved to a client's local machine. This can be saved in either XML or HTML format. Once a chart is plotted:
    a. Right-click on a chart once it is displayed in the web application, bringing up a pop-up window with various options.
    b. On the top left of the box select >file>Save as XML (or Save as HTML). This will pop-up a window that allows the user to select a location and file name to save the JClass markup to the client's machine.

    **Warning/caveat – As of 5/2011, a bug was discovered where loading a saved XML/HTML file into the chart GUI and then selecting a different number of forecast sources and re-submitting the form options to the VWT prevents updated plots from being created. This is related to a bug involving the code used to remove JClass markers (plotting points and lines) in PlotChart.java. A Trac bug ticket for this defect has been submitted (#649). Therefore, I would have users disregard this functionality

3. Saving the data of the chart– The data represented in the chart can be saved as an ASCII flat file (.dat) to their machine with the data in array format. The JClass chart software has a built-in ability for this. Once a chart is plotted:
    a. Right-click on a chart once it is displayed on the applet. This will bring up a pop-up window with various options.
    b. Select the “Data View” tab, then the “General” tab below it.
    c. At the bottom of this box is a “Save” button with the icon of a floppy disk. If you select the save button, a pop-up window will pop-up a window that will allow the user to select/choose a file name and location on the client’s machine to save the data to.
    d. Click the “save” button to save the file.

    By default a “.dat” extension is included for the file name. The arrays of information in this file should all have the same length. The format of this file contains the following information with an example of what a line of the information would like in the saved file :

    - Brief header explaining contents of the file.
        - # The data is in array format
        - # It has one set of x-axis data, multiple sets of y-axis data …
    - A line defining the dimension of the data array, not including the reference data
        - ARRAY ‘ ‘ 2 5
        - In this example, this means there were 2 forecast sources and 5 points of data.
    - A line with single-space delimited list of reference data (i.e. data or location names representing the data) with each value surrounded by single quotes.
        - ‘01/01/2009’ ‘01/02/2009’ ‘01/03/2009’ ‘01/04/2009’ ‘01/05/2009’
    - A line with single-space delimited of incremented integer values representing the number of points of data (with respect to one forecast source of data). The values start at “1.0” for the first index of data.
        - 1.0 2.0 3.0 4.0 5.0
    - A line for each of the forecast sources containing the name of the forecast source and the category type (separate categories or all categories) surrounded in single quotes and space-delimited data values. Each of the values in the line corresponds to the associated reference data and integer value representing the index of the array of data.
        - ‘manual All’ 31.08 22.97 18.18 NaN 31.82
        - ‘auto All’ 11.5 17.5 43.94 -1.52 27.27
        - In this case the forecast sources chosen were manual and auto forecasts and the total (All) categories combined were used.

Command Line Version

Overview

The verification tool has the ability to be run by command-line initiation in addition to the web application. Since this does not involve manual interaction through a web application, but rather the processing of submitting settings by a settings XML file, this is often referred to as a 'static' version of the verification tool, which may appear in some of the file names including the driver used in the command-line instantiation, 'StaticRunDriver.java'. This version allows a cron job to automatically create output with updated data routinely. This convention also appears in the downstream post-process software created by Adam Allgood and Scott Handel (CPC) that creates graphics using verification ASCII input. That process is called the 'Static Plots' process, which is a separate downstream process from the verification process but would utilize output from the command-line run version of the verification system process which could be considered 'static' since graphics are pre-processed prior to the users viewing output.

Instead of results being displayed on a web application, files with results are created in the 'output' subdirectory of the process. File names are automatically generated by the verification system process by using information about the settings submitted and the data retrieved to create the file names.

This version can be ran two ways :

1. Direct-access Database - Preferred. This allows the executables to access the MySQL database directly. Quicker run-time than the web service access version (listed below). The security utilized in the web service version would not typically be needed because the command-line version is typically ran internally, where direct access to MySQL would be secure.
2. Web service access Database - Not preferred, unless stricter IT security. This utilizes the Web service technology (SOAP services) to access MySQL during run-time. More secure, added level of security when accessing data from the database. Since this version has

Features

The command-line version of the running the verification software contains the same or more options as the web application version to produce output files. There are certain settings that would be preferable to be done by the command line run version than the web application version, especially where certain settings require large amounts of data to be processed which would take too long or crash the web application version.

Here is a list of features of the command line version of the verification software:

- Ability to process multiple runs of the verification software with one settings XML file.
- Output of PNG graphic files for charts, or flat ASCII files. There is no ability to create spatial map graphics for the command-line version in this verification system. However, the process mentioned before called 'Static Plots Process' created by Adam Allgood and Scott Handel can be ran downstream utilizing the ASCII output from this verification system. The Static Plots process can create contoured maps of scores from the ASCII output of this verification system (which cannot be done by the verification system itself at this time). Documentation of the Static Plots Process is available as a Google Doc in the Google Drive folder labeled 'Verif_staticPlots' under the 'Verification' folder : https://drive.google.com/a/noaa.gov/?tab=mo#folders/0B5iFqLMmodO7WTZaRlIzVDhWZFk
- A log file (/output/static.log) is produced during each run. The level of logging desired can be set in the log configuration file /input/logConfig.txt

Prerequisites

You must set the $VERIF_HOME global environment variable in your .cshrc. This is the root (main) path of where the verification software is located. Software must also have been properly compiled and built with necessary configuration files.

Usage

As outlined in this section's 'Overview' section, there are two ways of running - the implementation that accesses the database directly (preferred due to less run-time) or using web services to access the database (not preferred for command-line due to longer run-time). There are various Ant commands that can be issued on the command-line to compile, build the API documentation viewable in a browser, and run the process. The 'ant -p' command will list possible targets (commands). The main targets are listed below.

See the How to Run Document for details. Below are some general commands that you would use to run the software (after configuration and option settings set according to How To Run).

To run the command-line version of the verification tool by having the system access the database directly (preferred for command-line version) you can either use the Ant target command on command-line :

```
        > ant run_command_line_direct_access
```

or directly enter in a java command, for example :

```
>java -jar build/verif_client.jar -f input/settings.xml -l input/logConfig.txt
```

To run the command-line version of the verification tool by using web services (not preferred typically for run-time purposes and it is not as well tested) :

```
> ant run_command_line_web_services
```

These Ant commands initiate the process that begins by running the 'StaticRunDriver' class, which can be viewed as the initiation driver script. This then runs the main verification driver class that does much of the processing of the data and statistics, which is also called by the web application version as well.

High-level process flow for the direct-access database command-line version

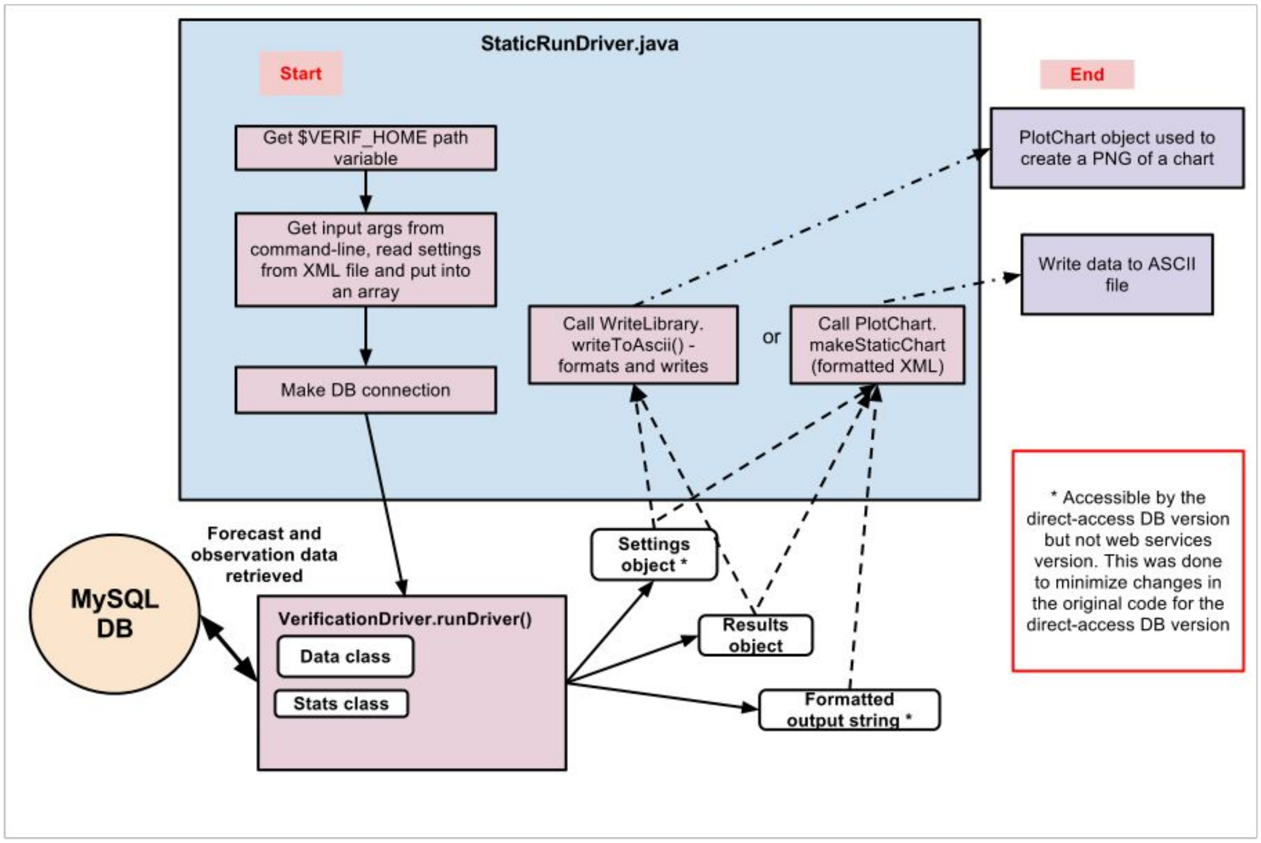

StaticRunDriver.java
Start
End
Get $VERIF_HOME path variable
Get input args from command-line, read settings from XML file and put into an array
Make DB connection
Call WriteLibrary. writeToAscii() - formats and writes
or
Call PlotChart. makeStaticChart (formatted XML)
PlotChart object used to create a PNG of a chart
Write data to ASCII file
* Accessible by the direct-access DB version but not web services version. This was done to minimize changes in the original code for the direct-access DB version
Forecast and observation data retrieved
MySQL DB
VerificationDriver.runDriver()
Data class
Stats class
Settings object *
Results object
Formatted output string *

Fig. 7: Process flow diagram of the direct-access database command-line run version of the verification tool.

Process flow

The method shares the same core driver (VerificationDriver.java) as the web application but differs in the way that the main driver is initiated and how output is created.

The steps of the overall process flow for the command line (static) processing is as follows:

1. The software can be initiated either by using a 'java' command on the command-line with all the necessary options submitted as arguments, or an Ant command can be used to kick off the process (ie. 'ant run_command_line_direct_access'). The 2 methods are described in detail in the 'How To Run' file. Below, both methods are briefly described.

   Ant build.xml file – The build.xml file tells the verification.jar file to use StaticRunDriver.java for command line initiation :

   ```
   <attribute name="Main-Class" value="gov/noaa/ncep/cpc/driver/StaticRunDriver" />
   ```

   Options are included in the command to set the name of the settings XML file and log4j log configuration file. When the command-line run version is initiated by running the Ant command, 'ant compile_command_line_web_services' , information in build.xml is used for run-time such as using the $VERIF_HOME global environment variable, the files to utilize for the settings XML options to run the process for, the configuration file for log4j, and information about where the .jar files are located.

   or

   java -jar... command-line initiation - Input arguments with the settings XML file and log configuration file must be specified and submitted. This is essentially what is ran when the Ant build file is used (above). This is done by using a command like this on the command-line (-f option for the settings file location and name, and -l option for the log4j log configuration file location and name):

   ```
   java -jar build/verif_client.jar -f input/settings.xml -l input/logConfig.txt
   ```

2. The main method in StaticRunDriver.java is initiated – The following steps occur in the main method:
   a. The value of the global environment variable $VERIF_HOME is retrieved from the .cshrc/.profile. The output from this process will be put in the output sub-directory of the main directory of the verification process. If this $VERIF_HOME is not found, a fatal error message is logged and the process is exited.
   b. Options regarding the settings XML and log configuration file are read in and applied. This is retrieved as command-line arguments, either directly submitted if running the job with a java command (java -jar ...) or the arguments set in the build.xml file if using the Ant command with the build file.
   c. log4J logging is initiated.
   d. The StaticRunDriver() constructor is instantiated:
      i. Database connection retrieved
      ii. Settings are put into an array which is read in from the settings XML file.
      iii. A new VerificationDriver.java object is created (called driverObj).
      iv. A database connection and the path of the main directory of the software (set by the global environment variable $VERIF_HOME) is set to the driver object.
      v. The core driver process is ran by calling

VerificationDriver.runDriver(settings). Details on the process flow of this driver is in the section Driver Process Flow.

vi. The settings and results object is retrieved from the driver object (driverObj).

vii. Appropriate formatting is applied to the results. Depending on the output type set in the settings, either an ASCII file or PNG graphic of a chart is created. Below outlines the steps in the process to create the output:

If the output type is “ascii” :

1. create the file name, which uses information from the settings and data object.
2. A method is called from the WriteLibrary.java format package (writeToAscii() method) passing the settings and results objects, and the file name. At this point the results and settings objects would already be loaded correctly with data from the driver already being ran (VerificationDriver.java).

If the output type is “chart” :

1. A new chart object (associated with PlotChart.java) is created. This class is responsible for the creation of JClass software charts.
2. An applet size is set (currently set to 660,500. This is hard-wired in the StaticRunDriver.java code in the constructor).
3. The settings and results object is set to the chart object.
4. Retrieve the XML string of results from the driver object.
5. Call makeStaticChart() method from PlotChart.java, passing the xmlString of verification data. This method creates a JClass chart on a content pane in a created applet in memory and saves the chart as a Portable Network Graphics (PNG) file to the process sub-directory /output.

viii. Finally, close connection to the database.

Input

Localized Files

1. /input/settings.xml (File name can be changed, if set properly)

Purpose : This settings XML file replaces users making options via the HTML form on the web application and submitting them to the verification tool. The settings XML file can be named anything since the name of the settings file is set on the command line when running the command line run version of the verification tool following the flag ‘-f ‘. The default file specified in the Ant build.xml is the one listed above /input/settings.xml. This file exists in the /input sub-directory already and can be used an example file to run the tool with. You can also edit the values of this file when you

want to run the process for just one set of settings. This contains only one set of settings but multiple sets of <settings>…</settings> can be included in the XML file for multiple runs. Settings.xml is in the 'input' sub-directory of the verification software is an example of this and is described next. This file is read in by the software and runs the process for these settings.See Appendix B for more information about the options for each setting and a settings dependency table.
Note: Use spaces, **not tabs** for indenting XML tags. Using tabs will sometimes lead to run-time problems that cause errors that are not obviously connected to this formatting issue.

Format type (.png, .dat(Ascii), etc.): XML
Format information/standards (Formatting of file, naming convention, headers, detailed information about format of data) : Below is an example. Note that if you want to have the process run for multiple settings, you can either create separate sets of settings or some settings allow a comma separated list of settings such as fcstSource and regions. To run the same settings for more than one forecast source, separate the list of forecast sources by commas embedded in the <fcstSources> tag.

If you list multiple regions in the <regions> tag (also comma separated list) and <outputDimension> is set to 'time', the verification scores are calculated by aggregating over the list of regions producing one combined score for each time step. If outputDimension is set to 'space', the verification scores are calculated by aggregating over the time steps for each region point.

The initial XML tag at the top for version and the data tags surrounding the settings tags are required.

The names of the tags in the settings match the variables used in the Settings object constructor. There is documentation regarding these settings in the Javadoc API for the Settings constructor.

Example of what the settings_short.xml could look like (this file may have different values depending on the latest changes made):

```
<?xml version="1.0" ?>
<data>
<!--~~~~~~~~~~~~~~~~~~~~~ D+8 Temp ~~~~~~~~~~~~~~~~~~~ →
<!-- OFF / Heidke / Last 30d  →
    <settings>
        <variable>temp</variable>
        <fcstSources>off</fcstSources>
        <leadTime>08d</leadTime>
        <aveWindow>05d</aveWindow>
        <datesValidType>dateRange</datesValidType>
        <datesValid>20090502, 20090601</datesValid>
        <regionType>climateRegion</regionType>
        <regions>All</regions>
        <spatialType>station</spatialType>
        <outputType>chart</outputType>
```

```
                <outputDimension>time</outputDimension>
                <scoreType>heidke</scoreType>
                <categoryType>separate</categoryType>
            </settings>
        </data>
```

2. input/logConfig.txt and input/logConfigApplet.txt

   Purpose : Configuration file for logging log4j processing for the static process (not the verification web tool as a web application). This configuration file only need to be read in to the process once, at the beginning (ie. the initiation code). See more information about this file in the logging section of documentation in this file.

   Format type : ASCII text file.

   Format information / standards : Syntax follows log4j convention.

3. input/verif_direct_access.conf - for direct-access database
   Create this file in the /input directory by copying the verif_direct_access.conf.example files and editing as necessary following the 'How to Install' document directions. This contains both the database and threshold settings, rather than having them split up into two conf files as in the above version. These are accessed when using direct-access to database in the command-line running.
   or

   input/verif_client.conf and input/verif_server.conf - for web services database access
   Create these files in the /input directory by copying the verif_client.conf.example and verif_client.conf.example files and editing as necessary following the 'How to Install' document directions. Contains necessary settings such as for the database and QC thresholds, with each of them containing settings required by the client and server sides separately. These are accessed when using web services in either the web application or web services version of command-line running.

Operational input (centralized data sources):

File path/name: MySQL databases and tables. See the data description section in Appendix C. Inventory available at Verif_Datasets_Inventory Google Doc
Format type: MySQL database
Purpose: Provides the forecast, observation, climatology, and look up reference information.

Output

As mentioned in the overview of the static version of the web tool, there are two types of available output from running the command-line version of the web tool:

- ASCII file with columns of data with headers can be output

- JClass charts as PNG graphic files can be produced from running the static version of the VWT.

The ASCII can subsequently be ingested into a downstream process called 'Static Plots' which is external to this system to create contoured maps from gridded ASCII data from the verification system.

Output Location

All output is directed to the /output directory of the main directory of the verification software. The ASCII file names will have unique file names automatically selected based on various settings you run the process for. Logs are automatically written after processing to /logs/static.log

The file names of both the ASCII files and chart graphics are determined automatically based on various settings you run the process for. These programs call methods and pass the settings and data objects to FormatLibrary.java methods – getAsciiFileName() for the ASCII file, and getImageFileName() to build the name of the files based on settings and data information. The below section identifies the details of the formatting of the file names that are automatically generated of the files and details regarding the output..

For each set of settings, there is one ASCII file or chart created. Within those settings, some options allow multiple choices, such as forecast type. Therefore one ASCII file or chart graphic can contain results of multiple forecast sources (ie. manual forecast, NAEFS forecast, etc.). These forecast sources, however must have the same lead time, spatial type, and some other settings that must be the same amongst these sources.

File Naming Convention

The file name convention is as follows:
verif_$variable_$source_$lead_$aveWindow_$spatialType_$scoreType_$categoryLabel_$outputDimension_$datesValidType_$datesValid.txt (or with .png as extension for graphics)

where variables are denoted by a prepended dollar sign. Some more notes regarding this file name convention:

- DatesValidType and datesValid are purely based on the settings submitted.
- categoryLabel describes the type of category selected, assigned by SettingsHashLibrary.getGenericCategoryLabels.
- The datesValid piece of the file name uses the dates valid passed from the XML settings and converts commas to dashes ('-') and semi-colons to underscores ('_').
- The methods getAsciiFileName() and getImageFileName() in the FormatLibrary.java class in the format package builds the name of the files based on settings and data information for ASCII and graphic output, respectively.

Example:
For this type of graphic: Temperature, manual official revised fcst (manual, 0pt5m lead , 01m averaging window), gridded data format, verification period over 55 months, Heidke score, separate forecast categories assessed, time dimension used as output (values for each date), verify for January and July of 2001 and 2006.

File name would be:
(for total or separate categories ‘Cats’ is appended to the category type. Otherwise ‘Cat’ is appended to the category type).

verif_temp_manual_0pt5m_01m_grid2deg_55months_heidke_separateCategories_time_selectMonthsYears_01-06_2001-2006.txt

ASCII file output

The ASCII files are space delimited. Location names that contain spaces originally are formatted in the ASCII files with underscores instead of spaces for ease of parsing for users.

ASCII Header Content

The ASCII file output contains columns of data representing the resultant skill scores. There are some slight differences in the contents of the header depending on the settings. The header includes information about the following (The label used in the header is listed first below. These labels are followed by a semi-colon in the header in the file):

- ‘Content’ - Brief content description line (currently set to “Verification statistics ASCII data file).
- ‘Variable verified’ - Variable chosen in the submitted settings (ie. temperature, precipitation)
- ‘Score calculated’ – Score type chosen (ie. heidke, rpss)
- ‘Forecast type(s) verified’ – Type(s) of forecast verified (ie. manual, CFS, etc.)
- ‘Each forecast represents a’ - Length of time that the forecast type represents, this is not the length of time that the verification is processed for. (ie. 03m for a 3 month forecast)
- ‘Verification period’ – Length of time that the verification was calculated for (ie. 11 months, etc.), based on retrieved results of forecast dates in the database that were used in the dataset.
- ‘From’ - This is the time range that the verification represents. The start and end dates of the verification period is used to determine this (ie. 01/2005 to 12/2007 for a seasonal forecast)
- ‘Lead time of the forecasts verified’ – For the back end code, lead time is from when the forecast was issued to the center of the valid date (ie. 2m for a lead 1 seasonal forecast). For the user interface and plot titles, lead time is from when the forecast was issued to the start of the valid date (ie. 0.5m for a lead 1 seasonal forecast).
- ‘Region type of input data’ – Type of regions used in the forecast (ie. climate regions, climate divisions)
- ‘Regions included in verification’ – List of regions used in verification processing (ie. NE,SE,MW,S,HP,W)

- ‘Type of spatial formatting of the input data used in verification’ – How the forecast and observation data is spatially formatted (ie. gridded, stations).
- ‘Category type label’ - Type of category(ies) (scores including all categories). Labels assigned to this header by SettingsHashLIbrary.getCategoryLabel() which would be different for each categoryType and forecast source.
- ‘Output dimension’ - Reference output dimension, ie. scores for each location, time step, or probability bin (ie. reliability score).
- ‘Type of date filtering’ - Type of date stratification (ie. range of dates, select months, etc.)
- ‘Date filtering options selected’ - Date options for stratification

Note: The below 2 header pieces may or may not be included in the header depending on the settings (ie. type of score). These pieces of header information are impacted by the threshold QC settings in the configuration .conf files.

- ‘Percent of valid scores for $forecastSource forecast $categoryType category’ (for total category only) - where $forecastSource is the forecast source selected and $categoryType is the type of categories. This is the percentage of scores in the file that are valid (non ‘NaN’ values). There may be multiple lines if multiple forecast sources or category types are listed.
- ‘Number of valid scores for $forecastSource forecast $categoryType category’ - where $forecastSource is the forecast source selected and $categoryType is the type of categories. This is the number of valid scores (considering each row of data in the file) in the ASCII file itself (non ‘NaN’ values). For specific category verification the expected number of scores is not possible to calculate so this information would not be included in the header.
- Score summary - Includes the average score(s) for all forecast sources and categories and the number of values used to calculate the average score.

ASCII Columns Content

Data of the columns and the column headers of the ASCII file is as follows (in order of columns from left to right in the ASCII file):
In general the labels for the columns are underscore delimited.

- Reference array – Contains reference data for each score. The data and header name depends on the type of output dimension (space, time, probability). There are 3 types of output dimensions. These output dimensions and the associated header name, and type of data are as follows:

Output dimension	Header name	Data in column
time	Date_Valid	Valid date of forecast*
space	id	Location names**
probability	Probability	Probability bin range

* The format of the date depends on whether the forecast is short-range or long-range. This formatting of either MM/dd/yyyy (short-range) or MM/yyyy (long-range) is done in the data package during processing. This is determined by using the lead time unit in the passed lead time (ie. 03m, 08d). The lead time unit is the last character in the lead time passed and is either “d” (short-range) or “m” (long-range). The date is expressed as the middle date of the

valid period that the forecast represents.
** The format of space is as follows for the various *'spatialType'* settings (how the input data is formatted):
station - WMO station names. These names contain both underscores and spaces.
climate division - Name of climate division. Names contain spaces.
gridded - Position of the grid point formatted as XXYY where XX and YY are zero-padded integers representing a grid box of the X and Y positions, respectively. The bottom left grid point is 0101 and the top right grid point is 3619. The lat and lon represents the center point of the grid box. For example, a value of '*0411'* indicates that the X position is 04 and the Y position is 11. The associated lat and lon would be listed as 40 and -124, respectively.

The label for output dimension is determined by passing the output dimension from the setting to a SettingsHashLibrary.java method, *getOutputDimensionLabel()*. This method contains hash libraries with associated labels for each of the output dimensions.

- **Latitude and longitude** - If the output dimension is space, the latitude with column label **'lat**' is in the second column and longitude, labeled '**lon**' is in the third column. Otherwise, the second column is forecast source (see next bullet point). Each of these have field lengths of**.**
- **Forecast source** - This is the forecast source (ie. manual, auto, etc.). This is also in other columns preceding the column(s) of score(s) for that forecast source if there is more than one forecast source selected during run-time. The label for this column is '**Forecast_Source**'.
- **Verification score values** - There is some slight variation of the next few columns, depending on the settings. In general, a score would be listed next, and the following column would be either the number of valid forecasts that made up that score value, or the percent of valid locations or dates of the forecast data that made up the score. Number of valid forecasts are typically used for reliability (because you would not have an expected number of forecasts for each probability bin) and percent of valid dates or locations from the forecast data for all other scores. Some examples of data are :
  - total categories: '**all_cats**' would be the label for the column of scores, followed by the column for the percent valid dates/locations or number of valid forecasts making up the score.
  - separate categories: Each of the 3 categories' scores would be followed by either the number of valid forecasts or percent valid dates/locations. So for example, the headers would be in the order of '**B**, '**Num_Valid_Forecasts**', '**N**', '**Num_Valid_Forecasts**', '**A**','**Num_Valid_Forecasts**'.
  - a selected category: The scores for a category with the category used as the column label (ie '**above_normal**') followed by either the number of valid forecasts or percent valid dates/locations
- **Data QC info** - The number of valid forecasts would be labeled as either 'Num_Valid_Forecasts' and the percent of valid locations and dates would be labeled as 'Percent_Valid_Locations' or 'Percent_Valid_Dates', respectively. The dimension of percent valid forecast-observation pairs of data making up each score is the other dimension not represented by the reference data in the first column. For example if the selected output dimension was 'time', then the first column should contain dates, and the percent of valid locations would be included in each row of data (labeled 'Percent_Valid_Locations'). For some cases, such as for reliability (scores where

output dimension is ‘probability’), the column would contain the number of valid forecast-observations pairs in total that went into the score. This would be labeled as ‘Num_Valid_Forecasts’.

Field lengths of columns

Below the columns of data and the associated field lengths in bold square brackets are listed (in order of columns left to right).

- Reference data for output dimension:
    - space - **[28]**
    - time - **[15]**
    - probability - **[15]**
- lat * **[8]**
- lon * **[8]**
- Forecast source - Either the length of the forecast source, or the number of characters in ‘Forecast_Source’, whichever is greater.
- Score - Either the length of the category label or default character length assigned as variable defaultCategoryLabelLength in WriteLibrary.java, whichever is greater.
- QC information (# or % valid fcst-obs pairs) **[25]**

* Only printed for spatial output dimension output. See previous section for more info.

Impacts of QC on ASCII file output

There are 2 types of QC that impacts the ASCII file output. See the Quality Control Documentation section for details. This section outlines situations where score values are set to ‘NaN’ based on QC. The Google Doc ‘Verif QC’ in the ‘Verification System’ Google folder is also useful : https://docs.google.com/a/noaa.gov/document/d/1kVTfRg-sxzSjTFlciB3elMSoCDHrgc2aC11vZ71vym8/edit#.

Chart graphic output

The graphic that is saved from running the static version of the verification tool on the command line looks identical to the chart that would result from running the web application version. In general, the x-axis represents the reference information, typically valid dates, and the y-axis represents the values of the scores.

A legend is plotted underneath the plot. This represents all information plotted on the chart. Colors in the plotting package were selected so that if there are separate categories, the warm colors represent the above-normal categories, the cool colors represent the below-normal categories, and the yellow and neutral represent the normal categories. If total categories were selected, the order of colors representing the different forecast sources will use the same order with no meaning attached to the colors.

Reference data associated with the scores

Reference data is referred to as the data that represents the output dimension that the scores represent, such as the dates, locations, and probability bin values associated with the output score data. The type of reference data differs based on the output dimension setting. Possible values of output dimensions are time, space, and probability. The reference data is stored in an array that can be retrieved from the data or results object.

How the reference data is created

The reference data is produced in the Data.java in the data package. The loadData() method must first load the forecast and observation successfully. Once this is done, this method has if statements for each of the 3 output dimension types (time, space, and probability) and the associated reference data needed for each of these are created in different ways. In each of these if statements, the array variable 'referenceArray' is set equal to different reference arrays depending on the output dimension being processed, and retrieving the reference array will return the appropriate array.

It should be noted that reference arrays have unique values. For example, even though forecast data might have lots of data points for one date due to multiple locations of data, the reference arrays are unique, with no repeated dates. Also, reference dates are always saved into an array that is accessible by a data object (Data.java) get method, ie. getFormattedReferenceDatesArray() regardless of what type of output dimension.

How to retrieve/get the reference data

Once the data object is properly loaded with forecast and observation data and the *loadData()* method has ran in Data.java the method *getReferenceArray()* method can be called. Once the results object has been properly loaded, the reference array may also be retrieved from the results object by the same named method *getReferenceArray()*. The web services version must utilize the results object to retrieve this array (since the data object is inaccessible by the client's side of processing) , while the 'static' or command-line direct access database version could access the data object. However, both the web services and direct-access database versions utilize the results object for consistency. The verification system automatically utilizes methods to retrieve this, but since this is a relatively important feature, it is described in this section.

Reference data content

As discussed previously in the Data.java documentation section, loadData() method in Data.java sets the reference array to the correct reference data. There are if statements in the loadData() method based on values of outputDimension that decide what reference data to retrieve, either from a reference table in a database, the retrieved reference dates array, or a static reference array set in the class Stats.java. All arrays are of type one-dimensional String array. The current available types of reference data are:

- outputDimension = "time" : The reference array is set to formatted dates. Retrieved from a database query called by loadObsData() and formatted as well depending on the lead time unit (MM/yyyy for long-lead forecasts, MM/dd/yyyy for all other forecasts).
- outputDimension = "space" : The reference array is set to the location names. 4 different arrays are attempted to be retrieved – location names, longitude-latitude, latitude, and longitude. These are all obtained by calling methods in Database.java

that retrieve this information from reference tables in the database. Even though the reference array is set to the location name, the other location arrays are set so that they can be used for writing XML, for Google Earth plots.

- outputDimension - "probability" : The reference array is set to probabilities representing probabilistic forecasts. This array is retrieved from the Stats.java class. The method in this class, getProbabilityBinLabels(), returns an array of values that correspond to the bin labels ("0-0.1",..."0.2-0.33333",....."0.1-1.0"). These values must correspond to the other probability bin variables set in the Stats class (probabilityBinLowerThreshold, probabilityBinUpperThreshold, probabilityBinAxisLabels, and bins). See the Constants section of the Stats package in the next section for more information.
- category labels - The majority of the code uses hard-coded references to assigned category names (ie. "B", "N", "A"). These are set at the beginning of run-time via SettingsHashLibrary.java. This is called when?? To overwrite this, best to just do a cosmetic change in the display, ie. overriding a displayed legend label.

Driver process flow

Below are the process steps in the verification driver (VerificationDriver.java). The driver does not produce graphics or returns any score results output directly. It is left up to the application initiating the driver is responsible for retrieving the output in a desired format. However, the runDriver() method in the VerificationDriver.java does create formatted output, which can be retrieved by get method getXMLString() which is done by the command-line direct-access to database version of initiation. Web services calls this formatting separately, ie. within the applet. Even though the applet formats the scores the same way VerificationDriver.java does, the web services works requires processing to be handled differently.

In the driver constructor :

1. The log4j Logger is initialized. Configuration of the logger is not in the driver itself but in the application initializing the driver.

Steps in the main driver method

The main() method is only initialized if ran in the direct-access database command-line version. Otherwise, runDriver() is ran only.

1. The configuration settings from the verification system configuration file, /input/verif_direct_access.conf is read in .
2. Database connection is made.
3. The run-time settings XML file is parsed and read in.
4. A verification driver object, driverObj is created and calls runDriver(), passing the settings to the method.

Steps in the runDriver() method

1. Create new settings object.

2. Set the variable representing the path of the home directory of the process. The default null value is only overwritten for the command line run-time application (StaticRunDriver.class). The applet ignores this.
3. The format is set for how the output dates should look like on charts and ASCII files.
4. MySQL settings get loaded from the configuration .conf files.
5. The getResults() method (in VerificationDriver.java) is called, passing the database connection and settings object. getResults() then does the following :
    a. A data object is created, with the database connection and a date format passed.
    b. The Data class method to load the data into the data object is called (loadData()). Detailed documentation of how the data package works is in Appendix C.
    c. A stats (statistics) object is created.
    d. The Stats class method to calculate the statistics (scores) is called (calcStats()).
    e. Various set methods are called to set information in the results object, such as reference information and statistics from the stats and data objects.
    f. A Results object is returned, with all the necessary score and reference data loaded.
6. The array of score (stats) values are retrieved from the returned results object.
7. The stats are formatted appropriately for the desired output. This results in a formatted String of scores. get() methods in the VerificationDriver.java class enable outside classes to now retrieve the formatted string of score data.

Transmission Information

Nothing from this process is directly transmitted to a centralized data area. Typically, another process referred to as the 'static plots' process (See Compute Farm RFC #253) initiates this verification system on cron operationally on Compute Farm to produce ASCII files that gets processed by the static plots process. That process creates graphics such as contour maps, timeseries charts, etc. that get displayed on the CPC Verification Summary page.

Bugs & Issues

- Web service command-line version does not work - gives errors.
- See associated Trac tickets for specific issues or work to be done : Verification Trac Tickets
- Many of the common issues arise from the data itself - forecast and observations data. This can arise from either problems with the original ASCII files of data provided by various data owners (see data inventory link in the Resources section). Also, there could have been incorrect settings or failure of the data import Perl scripts running. On the IT side, sometimes replication of data between machines also leads to problems in data. Specifically, database issues typically include, missing data, or duplicate data.

Future Work

- See associated Trac tickets for specific issues or work to be done : Verification Trac Tickets
- There is likely some unnecessary duplication of storing and retrieving information, such as the results object vs. settings and data information. This is because the results object was needed for web services, but Damian did not want to change a significant amount of code in the direct-access DB version.
- Results.java may be moved out of the services package and into another package, since it may be used for the direct-access DB version in addition to the web services version.

Appendix A: Web Application

See the VWT Web application README

Appendix B: Run-Time Settings

This section is intended to list and explain the settings which are variables that the software accepts. These settings were selected to allow the verification tool to accept many options and types of forecasts to verify for. This should be similar/identical to the available documentation in the Javadoc API for the Settings.java settings constructor method.

Settings Dependency Table

This table describes each setting's dependency. In other words, some settings change depending on what other settings are chosen. For each setting in the left column, the settings that will affect that setting are listed in the right column.

Setting	**Depends On**
Variable	None
Forecast Type	None
Lead Time	fcstType
Average Window	fcstType
Dates Valid Type	None
Dates Valid	fcstType, datesValidType
Spatial Type	fcstType
Region Type	spatialType
Regions	regionType
Output Type	None
Output Dimension	outputType, scoreType
Forecast Source	fcstType
Start Date	fcstType, datesValidType
End Date	fcstType, datesValidType
Score	None
Category	scoreType, outputType

These are the settings required for the static version:

	Description and Valid Settings Combinations	Possible values
variable	Variable to perform verification for.	temp precip Uneven terciles/extremes: *${variable}-${categoryUnit}-${thresholdLower}-and-${thresholdUpper} e.g. tmax-ptile-15-and-85, where categoryUnit can be degF, degC, Ptile, mm, and in*
fcstSources	List of forecast sources separated by commas. Logic downstream separates the forecast sources appropriately. Different sources are available for different forecast types. See /trunk/web/library/existingFcstSources.js for a list of all the database table names.	manual auto tool (cca,ocn)
leadTime	Time to the center of the valid period from the date issued for all the back end code and database. Format is ${num}${unit}, where ${num} is the number representing the lead time according to the following ${unit} unit of time. ${unit} can have the values of characters "d" and "m", for days and months, respectively. 0pt5m is the revised monthly, issued last day of month valid for following month. 01m is monthly forecast, issued mid-month for following month. 02m - 14m is seasonal forecast, issued mid-month for following 3 months. For 6-10day, leadTime must be 08d and aveWindow must be 05d. For 8-14day, leadTime must be 11d and aveWindow must be 07d.	08d 11d 0pt5m 01m 02m … 14m

	For monthly, leadTime must be 0pt5m or 01m and aveWindow must be 01m. For seasonal, leadTime must be 02m,03m,. . ., or 14m and aveWindow must be 03m. Note, for the user interface and plot titles, the lead times for monthly and seasonal are the time to the start of the valid period from the date issued, consistent with what is currently displayed on CPC’s official forecast maps. For monthly, leadTime is 0.5m for the official forecast and 0m for the revised forecast. for seasonal, leadTime starts at 0.5m and goes to 12.5m.	
aveWindow	Width of valid period. Format is ${num}${unit}, where ${num} is the number of time units (according to type ${unit}) that the verification covers. ${unit} can have the values of characters "d" and "m", for days and months, respectively. For 6-10day, aveWindow must be 05d and leadTime must be 08d. For 8-14day, aveWindow must be 07d and leadTime must be 11d. For monthly, aveWindow must be 01m and leadTime must be 0pt5m or 01m. For seasonal, aveWindow must be 03m and leadTime must be 02m,03m, . . ., or 14m.	05d 07d 01m 03m
datesValidType	Type of way to filter the data to verify for:	dateRange selectMonths selectMonthsYears selectSeasonalSignal

Filter method	value of datesValidType
Days in a range:	dateRange
Select months for all years:	selectMonths
Select months and select years:	selectMonthsYears
Select seasonal signal	selectSeasonalSignal

	For charts, only dateRange is available via the web form. All forecast typesare valid for all of these options.	
datesValid	Formatted string of date information representing how to filter the data to verify for as specified by dateFilterRequestType. These dateValidRequest values applies to the valid date of forecast data.	YYYYMMDD,YYYYMMDD YYYYMM,YYYYMM

	The format allows for multiple combined ways to filter. The set within one type of filtering is comma delimited, the different sets of filtering specifications are separated by a semi-colon.:	MM,MM MM,MM;YYYY,YYYY ENSO;warm ENSO;neutral ENSO;cold

Forecasts valid for	datesValidType	datesValid
Days between 20090101 and 20090620	dateRange	20090101 0090620
January and July months for all years:	selectMonths	01,07
January and July months that are in the years 2000 and 2005	selectMonthsYears	01,07;200 2005
3-month season where there was a warm ENSO episode	selectSeasonalSignal	ENSO;wa rm

If outputType=chart, only dateRange for dateValidType is available via the web tool.
If dateValidType=dateRange and fcstType=6-10day or 8-14day, datesValid are in the format YYYYMMDD,YYYYMMDD, where the dates are the range of dates to be used.
If dateValidType=dateRange and fcstType=monthly or seasonal, datesValid are in the format YYYYMM,YYYYMM, where the dates are a range of months or seasons to be used.
If dateValidType=selectMonths, datesValid are in the format MM where MM is a comma separated list of months and all years available are used.
If dateValidType=selectMonthsYears, datesValid are in the format MM,MM;YYYY,YYYY, where MM and YYYY are a list of months and years to be used.
If dateValidType=selectSeasonalSignal, datesValid can be ENSO;warm, ENSO;neutral; or ENSO;cold. For information on how these ENSO events are classified see the Seasonal Climate Phenomena section of Appendix C.

regionType	Type of regional division to use for verification calculations.	climateRegion state

		climateDivision
regions	Particular region(s) to perform verification for. If regionType=climateRegion, valid regions are: W, HP, NE, SE,MW,S, or All. If regionType=state, valid regions are a comma separated list of 2 letter state abbreviations, or All. If regionType=climateDivision, valid regions are a comma separated list of climate division numbers (1-102) or All.	NE,SE,W, All NY,NJ,CT,RI 8,16,98
spatialType	Type of storage of each data point. The datasets of forecasts and/or observations are made up of different types of ways that the data represents. If fcstType=6-10day or 8-14day, spatialType can be station or gridded. If fcstType=monthly or seasonal, spatialType can be gridded or climateDivision.	station gridded climateDivision
outputType	Method of displaying and/or producing output for a score. This is the representational format of the output data produced by the verification software. Map is not valid if scoreType is reliability. If outputType=chart, dateValidType must be dateRange.	chart map ascii
outputDimension	Dimension that the output represents. It is the reference dimension that each of the score values represent. Examples of output dimension: Time is for scores represented by dates on a timeseries chart. Space is for scores represented by spatial points on a map. Probability is for scores represented by probability values for reliability scores where the scores are for an interval of probabilities for a forecast category. If outputType=chart and scoreType is not reliability, dimension must be time. If outputType=chart and scoreType=reliability, dimension must be probability. If outputType=map, dimension must be space. If outputType=ascii and scoreType=reliability,	time space probability

	dimension must be probability.	
scoreType	Type of verification score to calculate. If scoreType=rpss, categoryType must be total. If scoreType=reliability, outputType must be chart or ascii. If scoreType=reliability, outputDimension must be probability.	heidke rpss brier reliability
categoryType	How to plot scores for the categories. If scoreType=rpss, separate is not valid. If outputType=map, category must be either total, B, N, or A.	total separate B N A
ecType	Whether to include forecast of "equal chances" in the verification or not. See setEcType() method in this class. Note, for long range, only fcst sources on the EC list are verified using EC. For other fcst sources, EC forecasts are below the skill mask and are should not be verified. The list is at settingsObj.getLongRangeEcFcstSourceListArray().	noEC withEC default

These are additional settings for the web version:

	Description and Valid Settings Combinations	Possible Values
fcstType	A combination of leadTime and aveWindow is used by the back end code to set fcstType. For 6-10day, leadTime=08d, aveWindow=05d, spatialType=station or gridded. For 8-14day, leadTime=11d, aveWindow=07d, spatialType=station or gridded. For monthly, leadTime=0pt5m or 01m, aveWindow=01m, spatialType=gridded or climateDivision. For seasonal, leadTime=02m,03m, . . ., or 14m, aveWindow=03m, spatialType=gridded or climateDivison.	6-10day 8-14day monthly seasonal
Start Date	The back end code converts start date to	YYYYMMDD

	datesValid based on datesValidType, which depends on fcstType.	YYYYMM
End date	The back end code converts start date to datesValid based on datesValidType, which depends on fcstType.	YYYYMMDD YYYYMM

There are settings that are automatically set within methods based on a settings value passed and are not included in the passed settings array:

	Description	Possible Values
spatialTypeTableName	String piece of database table name representing the spatial type. See getSpatialTypeTableName() method API documentation in this class for more information.	stn grid2deg cd
numProbLevels	The number of probability levels as an integer. This is hard-wired currently in this class as '3'.	3
homeDir	Home directory of the verification system.	
dateRangeFormat	String format representing the date values as specified in datesValid. This representation is in format of Java SimpleDateFormat. There is a get and set method to access this information in this class.	

Fig. 10: Table with settings and variables used in the settings.

Adding a new settings option

To add a new settings option, follow code changes required below for system to take it into account. If the below steps aren't done, but a new option is set in the settings.xml during run-time, that option would just be ignored and would never be pulled in during run-time.

1. In format/XMLReader.java add the new option name (what it would be labeled as in the settings.xml tag) to the *orderedEleArray()* String array specified at the top of the class. The new option can be put anywhere in order in the current array of hard-wired options, **as long as the next step specifying the array index matches the corresponding index in this array.**

2. Declare new option type at the top of the Settings class in Settings.java as '*private String newOption*'.

3. In Settings.java:
    a. Add new option into the array index in Settings.splitStringArray(). These are specified at the top of this method. ie. *ECType = array[13];*
    b. The index of the array the new option is set to **must match the index that it was added to in step 1** (In XMLReader.java, as specified in the array variable *orderedEleArray*). So, for example, if you added the new option as the first one listed in *orderedEleArray* in XMLReader.java, it must be specified as '*newOption = array[0]*' in Settings.splitStringArray().
    c. Add this to the JavaDoc API documentation in the method documentation above the Settings class in the appropriate order. Include typical options in the documentation.
    d. Add a get and set method in Settings.java for the new option (ie. public String getNewOption() and public void setNewOption()).
    e. Add a line to method *printSettings()* to print the new option.
4. Now you can use the new option anywhere in the code. Make sure that in the log file during run-time, you can see the new option being ingested into the process (prints from the StaticRunDriver and/or VerificationDriver
5. Optional- Add logic to *SettingsHashLibrary.getSettingsValidationHash()* which performs quality control on the value of the settings option. If you do not have any constraints on this option, do not add any logic. If you do want to constrain the type of format or only allow specific values for the option, include logic in the associate *if* statement for the new option. Put the logic statement in the associated order that you set the option in the first step. The order will not necessarily break the code, but should be followed for easy to understand code.
    a. For example, to just set the only allowed possible values for a settings option (example for option called 'datesValidType':

```
//----------------------------------------------------------
// datesValidType
//
settingsValidationHash.put("datesValidType", new String[] {
"dateRange|selectMonths|selectMonthsYears|selectSeasonalSignal|selectMonthlySignal
",""});
```

where the options are separated by pipes in the '*put*' statement.

Alternative

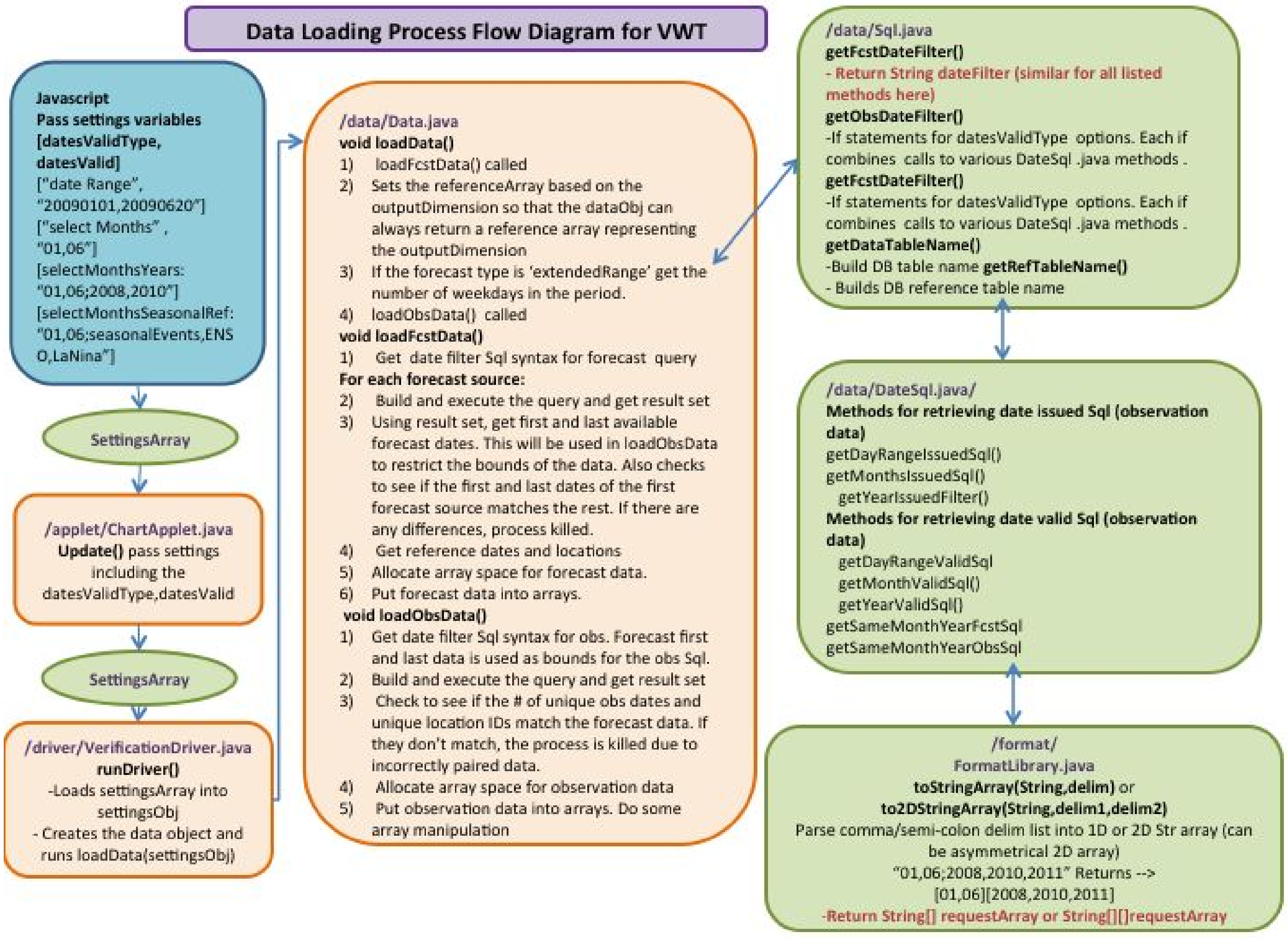
If you need to add a settings option, but cannot/do not want do it using the settings array (which is required to add in the settings passed by the Javascript/applet (web) and the settings.xml file):

1. Add a *get/set* method in the settings object for the new option
2. In *VerificationDriver.runDriver()* add the set method from the first step after the creation of the settings object.

An example of this being done is if the setting would not be typically part of the settings set by the user, ie. it should be determined other ways hidden to the user settings.

Appendix C : Data Package

This section is intended to explain how data is retrieved from the database. The functionality of retrieving data is contained in the data package (gov/noaa/ncep/cpc/data).



Data Loading Process Flow Diagram for VWT

Javascript
Pass settings variables
[datesValidType,
datesValid]
["date Range",
"20090101,20090620"]
["select Months" ,
"01,06"]
[selectMonthsYears:
"01,06;2008,2010"]
[selectMonthsSeasonalRef:
"01,06;seasonalEvents,ENSO,LaNina"]

SettingsArray

/applet/ChartApplet.java
Update() pass settings including the datesValidType,datesValid

SettingsArray

/driver/VerificationDriver.java
runDriver()
-Loads settingsArray into settingsObj
- Creates the data object and runs loadData(settingsObj)

/data/Data.java
void loadData()
1) loadFcstData() called
2) Sets the referenceArray based on the outputDimension so that the dataObj can always return a reference array representing the outputDimension
3) If the forecast type is 'extendedRange' get the number of weekdays in the period.
4) loadObsData() called

void loadFcstData()
1) Get date filter Sql syntax for forecast query

For each forecast source:
2) Build and execute the query and get result set
3) Using result set, get first and last available forecast dates. This will be used in loadObsData to restrict the bounds of the data. Also checks to see if the first and last dates of the first forecast source matches the rest. If there are any differences, process killed.
4) Get reference dates and locations
5) Allocate array space for forecast data.
6) Put forecast data into arrays.

void loadObsData()
1) Get date filter Sql syntax for obs. Forecast first and last data is used as bounds for the obs Sql.
2) Build and execute the query and get result set
3) Check to see if the # of unique obs dates and unique location IDs match the forecast data. If they don't match, the process is killed due to incorrectly paired data.
4) Allocate array space for observation data
5) Put observation data into arrays. Do some array manipulation

/data/Sql.java
getFcstDateFilter()
- Return String dateFilter (similar for all listed methods here)
getObsDateFilter()
-If statements for datesValidType options. Each if combines calls to various DateSql .java methods .
getFcstDateFilter()
-If statements for datesValidType options. Each if combines calls to various DateSql .java methods .
getDataTableName()
-Build DB table name **getRefTableName()**
- Builds DB reference table name

/data/DateSql.java/
Methods for retrieving date issued Sql (observation data)
getDayRangeIssuedSql()
getMonthsIssuedSql()
getYearIssuedFilter()
Methods for retrieving date valid Sql (observation data)
getDayRangeValidSql
getMonthValidSql()
getYearValidSql()
getSameMonthYearFcstSql
getSameMonthYearObsSql

/format/
FormatLibrary.java
toStringArray(String,delim) or
to2DStringArray(String,delim1,delim2)
Parse comma/semi-colon delim list into 1D or 2D Str array (can be asymmetrical 2D array)
"01,06;2008,2010,2011" Returns -->
[01,06][2008,2010,2011]
-Return String[] requestArray or String[][]requestArray

Figure 8: Flow diagram of data retrieval from the database. The above does not contain all details nor does it list all the actual methods included in the classes shown.

Overview

Javascript/the front end web passes settings to the Java settings package. Based on these settings, various methods builds pieces of the Sql query used to allocate space for data arrays, set reference array information, and load appropriate forecast and observation data.

Important things to know about correct processing of data:

- The Data.loadFcstData() method must be successfully ran before loadObsData() because it does the job of finding the first and last available dates of forecast data.
- Method setSdfSourceFormat() must be called by the main verification driver just after the data object is created (dataObj.setSdfSourceFormat("yyyy-MM-dd")). This is a format valid in Java SimpleDateFormat (sdf) that represents the format of dates in the used database for verification. This is required because sdf uses the format of the original date to format to a new date format.

  Quality Control done in Data.java while loading data

- The first forecast source is utilized to find the date bounds of the data. The first and last unique dates returned from the database are used to ensures that the first and last dates of all the forecast sources are the same and used as the date bounds in the Sql query used to retrieve observation data. If any of the subsequent forecast sources in the run do not match that of the first forecast source, the process is killed. This is required because the forecast arrays with multiple sources must be symmetrical as well as the same length with observations to have correct pairs. The upstream Perl import data scripts would typically insert ‘NaN’ values (instead of nulls) for all dates to ensure that there is a value (even if it is ‘NaN’) for every required date and location for symmetrical arrays.
- If the number of unique forecast dates and locations of subsequent forecast sources during a run do not match that of the first forecast source (during a multiple source run), the process is killed.
- If no forecast dates or locations were found, then process killed. This is done by checking to see if the number of unique forecast dates and locations returned is less than 1.
- If no observation dates or locations were found, then process killed. This is done by checking to see if the number of unique forecast dates and locations returned is less than 1.
- If there is not the same number of unique dates and locations between the forecasts and observations, process is killed.
- Checks are done with the forecast data to ensure correctly paired forecasts and climos (when climos are needed).

Summary of purposes of the different data package classes

Data.java – Driver of data package. Performs calls to load the data and load reference data arrays.

Database.java – Communicates with the database. Performs actual functionality of querying the database. There is a series of methods to retrieve different types of information from the forecast and observation data from the database, such as getting the number of unique results, counting rows of results, etc. There are 2 types of ways that these functions can be achieved - creating different Sql syntax depending on the information you want retrieved and executing them by communicating with the database - or - Retrieving as much data in the forecast or observation data in the database from one main Sql query and executing it. The data is retrieved as a result set and various Java functions extract information from the result set. In the second method, there is much less database querying which leads to much more efficient processing, since communication with the database is pretty costly on time during run-time. Therefore, the latter method is currently used in the software. The methods that utilize the first way have been retained in Database.java for debugging purposes or future possible use but are currently not used during processing.

This section of methods has a header in the source code to separate them (in Database.java):

```
//-------------------------------------------------------------
// Most of the methods below are no longer typically used by the verification process
// due to the slowness of making multiple queries to the database
// but are kept in to use for debugging or general use
//-------------------------------------------------------------
```

Sql.java – Handles building of the Sql syntax to create database queries. Various methods use select pieces of information from the settings object to build parts of Sql queries.

DateSql.java – Handles building of Sql syntax related to date filtering.

ReferenceDatesCreator.java – Creates arrays of reference dates by using Java calendar functions to build the array from a start and end date (set by the settings object). *Deprecated because of its lack of flexibility but kept in case dynamic querying of dates from the database takes too long during run-time and could be used for date range date filtering. However, this class only accounts for situations where the type of date filtering is by date range.

Data Loading Process Flow

Processing of loading data assumes the settings are loaded properly in the settings object. Then, a main driver for the verification process would create the data object and run the loadData(settingsObj) method in Data.java. The Data class contains various get methods where the data object can be used to access loaded forecast, observation, and reference data (once properly loaded). It should be noted that the results object does not contain the forecast, observations, or climatology data.

Data.java is the driver for the data retrieval. The first method called once the data object is created is loadData(). loadData() uses logic to determine what data needs to be loaded. This is a generic loading method that calls other necessary methods to load data. Note that forecast data is loaded prior to observation data in this method because the bounding dates (first and last available forecast dates) are used to bound the observation data loaded. This prevents incorrectly paired data from being loaded and processed. See the individual loadFcstData(), loadObsData() documentation below for details. The only argument you need to pass to the loadData() method is the settings object (properly loaded with settings before passing).

The loadData() method performs the following steps :

1. Calls loadFcstData() - loads the forecast data.
2. During the running of loadFcstData, information such as the issued dates are retrieved. The issued dates are retrieved for each forecast source. However, if an error occurs, such as a forecast source not having matching beginning and end dates, the whole process is killed. Therefore the issued dates retrieved should be correct for all

forecast sources if all the data from the forecast sources have matching dates. It is possible that you can just retrieve the issue dates for the first forecast source since the process will die if the rest of the forecast sources do not match the first one. This would only be needed to be changed if leaving this in the loop for all forecast sources is causing significant increase in run-time.

3. Based on the output dimension type, reference information is loaded into the reference array, which can be accessed by method in this class getReferenceArray(). If dates are the reference array (output dimension is time), for lead time "m", the format is MM/YYYY. For all other lead times, the format is MM/dd/YYYY. A reference array of unique dates are always available if loadData() is succesfully processed. This can be accessed by get method getFormattedReferenceDatesArray().
4. If the forecast type is “extendedRange”, an extra step is done to retrieve the number of week days so that a number can be retrieved for the expected number of forecasts. In the case of extended range forecasts, the official is only issued on the weekdays so the expected number should reflect that. This way QC-ing based on the number of data points is appropriate and relative to the correct expected number for the official forecast downstream. This information is used later on in QC-ing.
5. Calls loadObsData() - loads the observation data, checks done to ensure data matches with loaded forecast data.
6. Calls loadClimData() - loads the percentage of dry locations (based on location and time), if Settings.getDryLocationCorrection is set to true. The conditions specifying whether the dry location correction is done or not are set within that method.

Developer’s note: In the future if climatology or other data needs to be loaded as well, a load method would need to be created in this class and then called by this method.

The loadFcstData() method performs the following steps :

1. Retrieve the date piece of the syntax.

For each forecast source:

2. Build sql query syntax. Various methods are called to build the final query including retrieving the date piece of the syntax.
3. The query is executed and the results set is retrieved. The database is only queried once for forecast data. The result set is used to retrieve information rather than multiple database queries to make processing more efficient.
4. If it is the first forecast source :
    a. The first and last forecast dates are retrieved from the result set and saved into an array. (Used later on for comparison to the other forecast sources if they exist and as observations data date bounds).
    b. The first and last forecast dates retrieved from the previous step are set as the date constrains. These dates are accessed by a get method getFcstDateValidBounds() later on by the loadObsData() method to restrict the beginning and end of dates of loaded observations. This ensures matched pairs of forecasts and observations.
    c. Arrays are allocated for the forecast data, based on the first forecast data, since QC checks are used to ensure that the data is symmetrical. Array space is allocated according to the number of forecast sources, unique number of forecast dates, unique locations,and number of forecast categories for the forecast probabilities, and forecast categories.

If it is not the first forecast source :

   a. The first and last forecast dates from the current forecast source is retrieved from the dataset. These values are compared to the first and last dates from the first forecast source. If these are not equal, then the process is killed and a fatal error is thrown.
   b. If the number of unique forecast dates and locations do not match the first forecast source, the process is killed.

5. Retrieve the valid dates from the result set (unique values retrieved).
6. Format and set the reference dates to the valid dates that were retrieved.
7. Retrieve the issued dates from the result set (unique values retrieved).
8. Format and set the issue dates to the valid dates that were retrieved.
9. Retrieve the number of unique forecast dates from the result set.
   a. If there are no forecast dates returned, process is killed.
10. Retrieve the number of unique location IDs from the result set.
    a. If there are no forecast locations returned, process is killed.
11. Retrieve the forecast probability data from the result set and load into arrays accessible by get methods. The forecast data is now "loaded" into the data object and can now be retrieved by a get method in this class.

The loadObsData() method performs the following steps :

1. Build sql query syntax. Various methods are called to build the final query. including retrieving the date piece of the syntax. The dates in the query are bound by the first and last dates retrieved from the loadFcstData() method which was previously run. This is to ensure that the observations pair with the forecasts.
2. The query is executed and the results set is retrieved. The database is only queried once for observation data. The result set is used to retrieve information rather than multiple database queries to make processing more efficient.
3. Retrieve the number of unique observation dates from the result set.
   a. If there are no observation dates returned, process is killed.
4. Retrieve the number of unique location IDs from the result set.
   a. If there are no observation dates returned, process is killed.
5. If the number of observation dates or locations IDs do not match the number of forecast dates or locations, the process is killed.
6. Allocate array space for the observation data using the unique # of dates and location IDs.
7. Retrieve the observation data from the result set and load into arrays accessible by get methods. The observation data is now "loaded" into the data object and can now be retrieved by a get method in this class.

The loadClimData() method performs the following steps :

The loadFcstData() must be called prior to this method because the beginning and end dates are used to bound the retrieved climatology data according to the forecast data range. Also, checks are done with the forecast data to ensure correctly paired forecasts and climos.

1. Build sql query syntax. Various methods are called to build the final query. including retrieving the date piece of the syntax.

2. The query is executed and the results set is retrieved. The database is only queried once for climatology data. The result set is used to retrieve information rather than multiple database queries to make processing more efficient.
3. Retrieve the number of unique climatology dates (corresponds to each observation) from the result set. If there are no climatology dates returned, process is killed.
4. Retrieve the number of unique location IDs from the result set. If there are no climatology locations returned, process is killed.
5. If the number of climatology dates or locations IDs do not match the number of forecast dates or locations, the process is killed.
6. Allocate array space for the climatology data using the unique # of dates and location IDs.
7. Retrieve the climatology data from the result set and load into arrays accessible by get methods.

Building the Sql queries to obtain data from the database
The loadFcstData(), loadObsData(), and loadClimData() methods in Data.java call various methods from different classes to build Sql syntax that is sent to the Database.java class to execute the database query.

When the log4j logging is set to "DEBUG" mode, you can always see the SQL printed in the /logs/static.log file after processing.

The load methods perform similar steps to build Sql queries. The following is a list of steps that all methods perform:

1. Set a list of columns to retrieve. These are the names of columns in existing databases and temporary columns within the SQL command itself.
2. Get date filter. This is the Sql syntax piece of the query that represents the date filtering selections passed to the settings object to perform verification calculations on. There is a method for the forecast and observation date filter separately (Sql.getFcstDateFilter, Sql.getObsDateFilter). This is because the Sql syntax for the forecast needs to use additional Sql syntax to figure out the valid date from the date issued, which represents the forecast dates. The lead time and lead time unit information is used in the Sql syntax building to build the appropriate syntax. See figure 3 (Date Filter Process Flow Diagram for VWT) for a diagram of the work flow).
3. Build the INNER JOIN portion of the query. This is the SQL piece that physically joins the tables together. When the datesValidType is "selectSeasonalSignal", there are 3 tables joined together, the forecast (or observation, or climatology) table is one, the location reference table is one, and the climate phenomena table is one. For all other datesValidTypes, the climate phenomena table is not included in the above list of tables.
4. Build the JOIN condition. This is the part of the SQL query that specifies the conditions under which the tables should be joined. When the datesValidType is not "selectSeasonalSignal", the tables are joined only where the ids of the forecast (or observation, or climatology) table and the location reference table match. For a datesValidType of "selectSeasonalSignal", there is an extra condition to match where the dates of the forecast (or observation, or climatology) table and the climate

phenomena table match.

5. Build the WHERE clause. This is the SQL piece that further filters down the data. The WHERE clause becomes a combination of the date filter and an SQL piece that selects only the desired locations across the domain. When the datesValidType is “selectSeasonalSignal”, there is an additional piece of SQL that filters based on the desired value of the climate phenomena.
6. Build the ORDER BY clause. This just sets the order of the data that gets returned by the database, and is consistent for all three data types (forecast, observation, and climatology).
7. Combine all SQL pieces into a single SQL query.

   Sql.java has if statements based on the datesValidType (dateRange, selectMonths, selectMonthsYears) that call the appropriate methods from DateSql.java. DateSql.java has various forecast and observation each have methods that create the Sql syntax snippets for the date filter. These pieces of Sql syntax are returned to Sql.java, sometimes these pieces are combined by Sql.java to build the date filter syntax. Then these pieces of date filters syntax are returned to Data.java where the completed Sql syntax is passed to the Database.java class to retrieve data from the database.

   Comma and semi-colon delimited Strings are used for the datesValid setting which are parsed according to the delimiter and the DateSql.java and Sql.java classes parse these datesValid settings to translate to appropriate Sql syntax. For example if the setting for datesValidType = selectMonthsYears, and datesValid = 01,02,2009,2010, data for 012009, 022009, 012010, and 022010 would be retrieved.

   The Sql syntax representing time units is retrieved by using the methods SettingsHashLibrary.getSqlTimeUnit(), which uses the lead time (in number format - instead of 0pt 5, 0.5 is used) and converts the single character unit in that setting to convert to appropriate Sql syntax (“m” -> “MONTH”).

3. In the loadFcstData() method, if datesValidType is set to “selectSeasonalSignal”, then the special columns are set for the syntax. Then a for loop is used to cycle through all the forecast sources to create the base Sql query and then wrap the base query with extra syntax if needed. Then the query is executed (base query or special wrapped query).

In the loadObsData() and loadClimData() method, the date bounds are retrieved (set from the loadFcstData() method results) and included in the base Sql query. Then if special syntax is needed, wrapper syntax is used with the base query.

Below is an example of base Sql syntax for a set of settings. This is for typical situations that do not involve using another reference table to stratify forecasts by (does not use any joined tables).:

Example Sql syntax **(Use spaces not tabs for indented tags)**:
For the following settings.xml:

```
<?xml version="1.0"?>
<data>
```

```
<settings>
	<variable>temp</variable>
	<fcstSources>manual</fcstSources>
	<leadTime>02m</leadTime>
	<aveWindow>03m</aveWindow>
	<datesValidType>dateRange</datesValidType>
	<datesValid>200901,200905</datesValid>
	<regionType>climateRegion</regionType>
	<regions>All</regions>
	<spatialType>gridded</spatialType>
	<outputType>ascii</outputType>
	<outputDimension>time</outputDimension>
	<scoreType>heidke</scoreType>
	<categoryType>separate</categoryType>
</settings>
</data>
```

The forecast data retrieval Sql syntax is built:

```
SELECT data.id, data.date_issued, DATE_ADD(date_issued, INTERVAL 02 MONTH) AS
fcstDateValid, prob_below, prob_normal, prob_above FROM
cpc_forecasts.temp_manual_02m_03m_grid2deg AS data INNER JOIN
cpc_reference.grid2deg AS locationList ON data.id=locationList.id WHERE
DATE_ADD(date_issued, INTERVAL 02 MONTH) BETWEEN '2009-01-15' AND '2009-05-15'
AND climateRegion RLIKE '(.*)' ORDER BY data.date_issued, data.id
```

The observation data retrieval Sql syntax is built:

```
SELECT data.id, data.date_valid, data.date_valid, category FROM
cpc_observations.temp_03m_grid2deg AS data INNER JOIN cpc_reference.grid2deg AS
locationList ON data.id=locationList.id WHERE date_valid BETWEEN '2009-01-15' AND
'2009-05-15' AND data.date_valid BETWEEN '2009-01-15' AND '2009-05-15' AND
climateRegion RLIKE '(.*)' ORDER BY data.date_valid, data.id
```

Adding a new dataset

If a new input forecast source and corresponding observations satisfy the following requirements then no code has to be altered.

Forecast/obs requirements:

- The input ASCII forecast files follow this convention:
    - https://cpc-devtools.ncep.noaa.gov/trac/projects/Verif_System/wiki/DataTeam/InputRequirements
- The new table name follows the set conventions
    - Database Design (CPC Google Doc)

If the ASCII file adheres to requirements, the VWT upstream Perl software for importing into MySQL should correctly import forecast and observation data (and possibly climatology) into the DB.

To run the VWT, the input/settings.xml file would contain the new associated settings, ie. the forecast source (which must align with the snippet of the associated table name).

Datasets without terciles

Non-date specific date filtering (Seasonal Climate Phenomena)

This refers currently to the seasonal climate signal date filtering. Currently ENSO seasonal and monthly AO signals are the only signal available in the VWT options, but other ones can be added.

By selecting seasonal signal, regardless of the time scale of forecasts selected (daily, monthly, seasonal), valid dates of these forecasts that occur in a 3-month season are selected to verify. There is a table in the reference database specifically for seasonal climate phenomena. All phenomena that is seasonal can be added to this table.

To stratify by strength of ENSO, Michelle L'Heureux (the Climate Prediction Center) suggested using the ONI Oceaning Nino Index data:
http://www.cpc.ncep.noaa.gov/data/indices/oni.ascii.txt which is the data of displayed table:
http://www.cpc.ncep.noaa.gov/products/analysis_monitoring/ensostuff/ensoyears.shtml
ENSO events in the table are characterized by a minimum of 5 consecutive seasons exceeding the warm or cold threshold. In order to characterize ENSO by season in real time for this verification, Michelle suggested analyzing the value for each season individually. So if the value is <=-0.5, that season is considered a cold event, and if the value is >=0.5, that season is considered a warm event. A list of the ENSO signal for each season is at http://cpc-devtools/trac/projects/Verif_System/attachment/ticket/696/ENSO.txt, which is created by the script, /scripts/db_import_climPhenom.pl.

The reference table is formatted such that:
```
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| date_valid | date        | YES  | MUL | NULL    |       |
| ENSO       | varchar(20) | YES  | MUL | NULL    |       |
| AO         | varchar(20) | YES  | MUL | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```
As mentioned in the previous paragraphs, when datesValidType is "selectSeasonalSignal", the SQL query has extra information. Below is an example of Sql syntax for a set of settings where datesValidType is "selectSeasonalSignal":

Example Sql syntax for seasonal ENSO date filtering:
For the following settings.xml:

```
<?xml version="1.0"?>
<data>
```

```
<settings>
<variable>temp</variable>
<fcstSources>manual</fcstSources>
<leadTime>02m</leadTime>
<aveWindow>03m</aveWindow>
<datesValidType>selectSeasonalSignal</datesValidType>
<datesValid>ENSO;warm</datesValid>
<regionType>climateRegion</regionType>
<regions>All</regions>
<spatialType>climateDivision</spatialType>
<outputType>ascii</outputType>
        <outputDimension>time</outputDimension>
        <scoreType>heidke</scoreType>
        <categoryType>separate</categoryType>
</settings>
</data>
```

The forecast data retrieval Sql syntax is built:

```
SELECT data.id, data.date_issued, DATE_ADD(date_issued, INTERVAL 02 MONTH) AS
fcstDateValid, prob_below, prob_normal, prob_above FROM
cpc_forecasts.precip_manual_02m_03m_cd AS data INNER JOIN cpc_reference.cd AS
locationList INNER JOIN cpc_reference.climatePhenomena_seasonal AS seasonalSignal ON
data.id=locationList.id AND DATE_ADD(date_issued, INTERVAL 02
MONTH)=seasonalSignal.date_valid WHERE ENSO='warm' AND climateRegion RLIKE '(.*)'
ORDER BY data.date_issued, data.id
```

The observation data retrieval Sql syntax is built:

```
SELECT data.id, data.date_valid, data.date_valid, category FROM
cpc_observations.precip_03m_cd AS data INNER JOIN cpc_reference.cd AS locationList
INNER JOIN cpc_reference.climatePhenomena_seasonal AS seasonalSignal ON
data.id=locationList.id AND data.date_valid=seasonalSignal.date_valid WHERE
ENSO='warm' AND climateRegion RLIKE '(.*)' ORDER BY data.date_valid, data.id
```

The dry location climatology data retrieval Sql syntax is built:

```
SELECT data.id, data.date_valid, percentDry FROM
cpc_climatologies.percentDry_05d_1971_2000_stn AS data INNER JOIN cpc_reference.stn
AS locationList ON data.id=locationList.id WHERE date_valid BETWEEN '2000-01-01' AND
'2000-12-31' AND climateRegion RLIKE '(.*)' ORDER BY data.date_valid, data.id
```

https://docs.google.com/a/noaa.gov/document/d/1EIQnKwNl0ZBzY4bOv9smHMcM0a4Zaiky5hnTd-iynD4/edit#heading=h.a1sgtgcvhhwz

Adding new non-date specific filtering

A pre-requisite is that this reference data to filter by (ie. seasonal ENSO) is added to the database *'cpc_reference'*. Follow directions for this at https://docs.google.com/a/noaa.gov/document/d/1EIQnKwNl0ZBzY4bOv9smHMcM0a4Zaiky5

hnTd-iynD4/edit#heading=h.a1sgtgcvhhwz
It should be noted that this reference data to filter by must have daily values, even if seasonal or monthly data to make the matching of daily data to filtered data more efficient. The data import scripts take care of this by setting daily vaules to the seasonal or monthly values.

To add a new filter that involves a new table in MySQL, ie. a new signal to aggregate by, follow below steps :

1. In SettingsHashLibrary.java *getSettingsValidationHash()* add the new filter type of ie. datesValidType to the hash array, code block **//datesValidType** :
   settingsValidationHash.put("datesValidType", new String[] {"dateRange|selectMonths|selectMonthsYears|selectSeasonalSignal|selectMonthlySignal Add filter type here *",""});*

   Also, add an *else if* statement adding the name of the new filter type to settings variable *datesValidType* in code block **// datesValid** :
   else if (settingsObj.getDatesValidType().equals("selectSeasonalSignal")) {

   Note that a sub-type within this does not need to be added, ie. 'ENSO' which falls within the 'selectSeasonalSignal' *datesValidType*.

2. In SettingsHashLibrary.java method ***getSignalTableName*** add the name of the new reference MySQL table as a hash variable:
   stringHash.put("$referenceType","$tableName");

   where $referenceType is the new available string assigned to filter by (ie. selectSeasonalSignal) and $tableName is the name of the table in the MySQL database containing the added reference data to filter by (ie. climatePhenomena_seasonal).

Follow the below in 3 methods in Data.java (Unless noted) - loadFcstData(), loadObsData(), and loadClimData():

1. In **only** the loadFcstData and loadObsData methods (not loadClimData) : Typically you would use the variable *datesValidType* as the new type of filtering, ie. *'selectSeasonalSignal'.* There can be sub-types with this, ie. ENSO, with options, ie. warm or cold. Since this is a non-date filter, the dateFilter in the new case must be set to empty. In code block ***// Determine $dateFilter*** add an 'OR' to the if statement that checks for the datesValidType equal to the new filter:
   // If the datesValidType is 'selectSeasonalSignal' set this to empty
   if ((datesValidType.compareToIgnoreCase("selectSeasonalSignal") == 0) || (datesValidType.compareToIgnoreCase("selectMonthlySignal") == 0) Add OR statement here) {
   dateFilter = "";
   }
2. Add query string for setting the table name for the reference data. Add an *if (else if since an if exists already for selectSeasonalSignal)* statement to the code block ***// Build the INNER JOIN* part of the query**. This logic should retrieve the reference table

name and create *innerJoinStr* which is the snippet of Sql syntax that uses an INNER JOIN MySQL command and assigns a temporary data variable, *signal* for reference which is used in *loadObsData* to join to later on. The same temporary data variable *signal* can be used in the added *if statement* (typically innerJoinStr can probably typically be the same in each of the if statements). The value of $datesValidType passed to the call to method *SettingsHashLibrary.getSignalTableName($datesValidType)* must match the hash variable that was set in a previous step for SettingsHashLibrary. Below is an example of an *if* statement:

```
 if (datesValidType.compareToIgnoreCase("selectMonthlySignal") == 0) {
          // Get the name of the climate phenomena table
refSignalTable =SettingsHashLibrary.getSignalTableName("selectMonthlySignal");
          // Add to the INNER JOIN
          innerJoinStr = innerJoinStr + " INNER JOIN " + refDBName + "." +
refSignalTable + " AS signal";
 }
```

3. Add an *if* or *OR* statement in the JOIN code block labeled '***// Build the JOIN condition ("ON") of the query***' which uses the temporary data variable (ie. *signal*) for the inner join condition string :

```
          if ((datesValidType.compareToIgnoreCase("selectSeasonalSignal") == 0) ||
(datesValidType.compareToIgnoreCase("selectMonthlySignal") == 0) Add OR
statement here) {
          innerJoinConditionStr = innerJoinConditionStr + " AND DATE_ADD(" +
dateName + ", INTERVAL " + leadTimeValue + " " + sqlTimeUnit +
")=signal.date_valid";
          }
```

4. In **only** the loadFcstData and loadObsData methods (not loadClimData) : Add an *OR* to the *if* statement that sets the *whereClauseStr* . There are code blocks in both load sections labeled *'**// Build the WHERE clause**'.* This contains logic that sets the Sql syntax for using reference data (ie. ENSO). For monthly and seasonal climate phenomena an *OR* was added because the same *where* clause string is used.

Formatted reference dates

Reference dates are always loaded (by the Data.loadFcstData() method), so that post-processing after calculation can access this for plotting, writing data to ASCII, etc. In the case that the set outputDimension is "time", the reference array is set to the formatted reference dates.

To retrieve the formatted reference dates array (reference dates are always formatted), once the above function is performed (formatReferenceDatesArray() method is ran), you can call getFormattedRefDatesArray. How this works, is that by the loadObsData() calling the void method formatReferenceDatesArray(), it provides values to an array (variable formattedRefDatesArray) that is accessible to all methods in Data.java. Once these values are created in the format method, the getFormattedRefDatesArray can access this.

Available methods that perform specific database functions
Beside retrieving specific columns of data from the database, ie. probabilities and categories of data from the database based on the built DB queries, there are methods that accept an Sql query (or snippet of Sql syntax) and add other Sql syntax to return information, such as return counts of resulting rows of Sql, selecting unique results, etc. As mentioned above, most of these are not used during run-time anymore, due to using the result set instead of multiple queries to the database.

The best way to see what type of functions are available is to look at the Javadoc API documentation for the methods in Database.java.

Appendix D: Statistics Package

This section is intended to explain how the data is manipulated to calculate the specified score over the specified dimension. This functionality is contained in the Stats package (gov/noaa/ncep/cpc/stats).

Overview

Settings for the score type, output type, and forecast type used to calculate the score. The Stats package reorganizes the data based on output type in order to calculate the score over the correct dimension. A 3 dimensional score array is generated containing scores by model, category, and time/location/probability.

Purposes of the different stats package classes

Stats.java – Driver of Stats package. Calculating the score assumes the settings are loaded properly in the settings object. Then, the main driver for the verification process creates a stats object and calls calcStats (dataObj,settingsObj) in Stats.java. The Stats class manipulates the data to calculate the score over the correct dimension, determines if EC forecasts need to be removed or not, and calls the correct score calculating method. There is a qc method that checks the resulting scores for low quality, which can be set to remove scores that are derived from too few data or contain a low percentage of good data. The Stats class contains a get method where the stats object can be used to get the score array (see section of score array info).

Detailed info for Stats:

Process Flow

1. Data is retrieved from the data class and reorganized depending on output type.
2. Output arrays are initialized (size depends on scoreType).
3. Array manipulation to calculate score over correct dimension.
4. EC forecasts are removed, if applicable. This means that forecast values are replaced as a ‘NaN’ value. There is logic used for QC1(score quality based on # fcst-obs data points) to prevent these EC forecasts that get replaced as NaN from being counted

mistakenly as missing values when QC1 is performed.
5. Scores are calculated (with qc for each point), rounded, and inserted into score array.
6. Score quality control for all scores on entire plot.

Array Manipulation

Get methods in the data class are used to get the forecast probabilities and categories and observed categories. The dimensions of the arrays depend on the output dimension from the settings class. If the output dimension is time or probability, the dimensions are how they are set up by the data class:

Forecast probability is a 4-d array where the dimensions are: model, time, location, probability. Forecast category is an array where the dimensions are: model, time, location. Observed category is an array where the dimensions are: time, location. If the output dimension is space, the time and location dimensions of the 3 arrays described are switched. This is done so that the same code can be used to calculate the score regardless of the dimension.

Then the arrays are broken down into smaller dimensions so the score can be calculated for each time, location, or probability and model. First, the code loops over each model (the first dimension) and extracts forecast probability and category data for the current model. This decreases the forecast arrays by 1 dimension (now 2-d for forecast category and 3-d for probability). If the score type is reliability, these arrays are used to calculate the score (since it calculates over time and location all at once), otherwise further break down is needed. The reliability score is inserted into the score array for each model.

For all other score types another loop is used to loop over the second dimension (time or location depending on the dimensions set by the output dimension detailed at the beginning of this section). Data for the current time or location is extracted so that the 2-d forecast and observed category arrays are broken down into 1-d arrays and the 3-d probability array is broken down into a 2-d array. These category arrays now contain data for one time (all locations for that time) or location (all times for that location), and the probability array also contains the probabilities for the 3 categories in the second dimension. The score is then calculated and inserted in the score array for the current model and time or location.

The variable *fcstCat3d, fcstCat2d, fcstProb4d, etc.* are different arrays that contain forecast data in *Stats.calcStats()* in either category or probability format (cat vs prob, respectively). An example of the dimensions depending on output dimension are as follows:

outputDimension is Time :

    fcstCat3d : [model][time][location]
    fcstProb4d : [model][time][location][category]

outputDimension is Space :

    fcstCat3d : [model][location][time]
    fcstProb4d : [model][location][time][category]

outputDimension is Probability:

fcstCat3d : [model][time][location]
fcstProb4d : [model][time][location][probability]

Equal Chances Forecast Category (EC) Handling

The equal chances category can either be included or not included in the score calculations. A variable called *ecType* is set to either *"withEC"* or "*noEC"* to either include or exclude equal chances (EC) category forecasts, respectively. When the 'noEC' option is on, the EC values are converted to "NaN" for calculation.

The inclusion of EC forecasts can change the resulting scores. There are some special methods that account for this depending on the score type. See 'Equal Chances Forecast Category (EC) Score Handling'.

Setting/Determining/Retrieving EC Type

The EC type can be specified in settings.xml in the tag 'ECType'. There are 3 possible values for EC type:

1. withEC
2. noEC
3. default

Here are the steps that occur in order to propagate the EC type from settings.xml to the code:

1. *XMLReader.java* processes *settings.xml*
2. The `runDriver()` method of *VerificationDriver.java* creates a settings object using the array of settings from `XMLReader.java`
3. `runDriver()` calls `settingsObj.setECType()`, passing it the EC type from *settings.xml*
4. `settingsObj.setECType()` uses the following logic to set EC type within Java:
    a. If ECType is "withEC" or "noEC", then it's simply set to that.
    b. If ECType is "default", then:
        i. If the given forecast is extended range (lead time unit is "d"), then it sets ECType to "noEC".
        ii. If the given long range forecast (lead time unit is "m") is to be scored using the Brier skill score, then ECType is set to "noEC", since the calculation of the Brier skill score requires each forecast to favor a "winning category" (where the forecast probability of one category occurring is higher than the other two) and EC forecasts have no winning category.
        iii. Otherwise, ECType is set to "withEC" (Heidke, RPSS, and reliability). There are no other lead time units besides "d" and "m" at this time and it is not expected that there would be in the future.

The value of ECType can be retrieved with the get method `getECType() from the` settings object.

Stats usage of EC type

The stats class calls different versions of the heidke and reliability methods (if heidke or reliability is the score type) depending on whether EC is withEC or noEC. StatsLibrary.java contains different versions of the methods that treat the scores differently. See 'Equal Chances Forecast Category (EC) Score Handling'.

The Stats class calls a method before score calculation and QC, *dataObj.getNumDataByCategory()* which returns the number of found forecast points in each category of below,normal, above, and equal chances. This must be done prior to the methods that replace any probability or category values of 0 with NaN (*QCLibrary.removeZeros)* to determine how many EC forecasts were made. This number needs to be used for QC later on (see section 'EC forecast impact on QC').

EC forecast removal for special cases

Forecast sources not on this EC list are not verified using EC regardless of ectype ("noEC"). Therefore, "EC" forecasts - forecasts with skill data (comes with the forecast data, this is like forecast skill or confidence) less than 0.30 are removed. This is because forecasts of EC are below the skill mask and should not be verified. Long range forecast sources not on the EC list contain EC forecasts but these forecasts are just forecasts that are below the skill mask and have probabilities of .3333 for all categories and a forecast category of 0. For Consolidation, a skill mask is already incorporated into the probabilities and EC forecasts should be verified (hence consolidation is on the ecList). The code currently only handles ec for long range. In the future, a separate list can be created for short range ec forecast sources, and the extended range stats qc section can be modified to work like the long range.

Score Array Info

There are two versions of the 3 dimensional score arrays stored in the Stats class. One is scoreCatFloatArray and the other is scoreCatStringArray. Both can be access by get methods. These arrays contain scores by model, time/location/probability, and category, where index 0 is the model, index 1 is category, and index 2 is time, location, or probability. The time is a date for extended range forecasts, or a month or season for long range forecasts. The category dimension has a length of 4 where index 0 is the score for all categories together, index 1 is Below, index 2 is Near normal, and index 3 is Above. Index 0 may or may not contain EC forecasts, depending on the forecast type and score type. See the Types of Verification section for how EC forecasts are handled.

Score Quality Control

The Stats portion of the process performs quality control, which determines whether to keep or replace score(s) with 'NaN' values. See the Quality Control section for more details.

If statements

These are hardwired if statements that pertain to particular settings. If a new setting is added that relates to the Stats class, new if statements may need to be added. If statements for outputDimension equals time, probability, or space are used to set the arrays with the correct dimensions. If the scoreType is reliability, an if statement uses the forecastType to call the appropriate get methods to get the reliability diagram probability bin variables. If statements check if forecastType is extendedRange or longRange and if ecType is noEC or withEC to do appropriate quality control on EC forecasts. If statements check scoreType to call the correct score calculation method in StatsLibrary.

Constants

In various get methods in the Stats class, the number of probability bins and the values for the probability bins for the reliability diagram are set. The variables set within their respective get methods are: bins, probabilityBinLowerThreshold, probabilityBinUpperThreshold, probabilityBinLables, and probabilityBinAxisLabels. There are three versions for each of the get methods for each of these variables. Get methods with the suffixes ExtRange and LongRange should be used if the program has not yet run through the section that sets the reliability diagram variables that depend on forecast type (near the begining of calcStats), otherwise the version of get methods without those suffixes should be used. It had to be done this way since the probability arrays contain an extra bin for EC forecasts, so they initially had to be declared with unique get methods (the ExtRange and LongRange suffix methods). Once they are set with these get methods near the beginning of calcStats, a generic version of each of the variables can be used. The arrays, probabilityBinLowerThreshold and probabilityBinUpperThreshold, are the same length as the variable, bins, and contain the lower and upper thresholds, respectively, of the probability bins. The array, probabilityBinAxisLabels, is used for plotting the tick marks on the plot and the length must be one longer than the number of bins. These values correspond to each probability bin threshold from the lowest threshold to the highest threshold. The array, probabilityBinLabels, contains a list of each bin's range of probabilities. It must be the same length as the variable bins, and it must match the lower and upper bin thresholds.

StatsLibrary.java

Contains all of the methods to calculate the different types of scores. There is also a method to determine the highest, lowest, and average score for each model (using methods in the MathLibrary class).

Detailed info for StatsLibrary

Types of verification

The types of verification scores available are: Heidke skill score, ranked probability skill score (RPSS), Brier skill score (Brier), and reliability diagram. Details about the equations can be found in "Statistical Methods in the Atmospheric Sciences" (Wilks 2006) and details about the methodology can also be found in the javadoc in the code. For monthly and seasonal manual forecasts, whether EC forecasts are verified or not is unique to each score type. The way that EC forecasts for long range manual forecasts are handled are: Heidke, RPSS, and reliability diagram scores contain EC forecasts, and Brier scores do not contain EC forecasts. Scores are generated for the Below, Near Normal, and Above categories and also for all categories together, where all categories together may or may not contain EC forecasts as described earlier in this section. If the manual forecast is being verified with EC forecasts, the scores for all categories together contain EC forecasts. Scores are not calculated for EC forecasts separately, since those scores would be a constant.

Heidke

There are several different Heidke methods that are used. There is also a method that takes two two-dimensional arrays and returns a single score, but this method is not used. For extended range forecasts, *calcHeidkeNoEc* does not do any dry location correction, and *calcHeidkeNoEcDryLocationCorrection* includes a dry location correction. See the next section, Heidke Dry Location Correction for more information on this. EC forecasts (zeros) should be removed from the forecast array before using the noEC versions of *calcHeidke*. For monthly and seasonal forecasts, *calcHeidkeWithEC* is used. The reasoning behind the handling of EC is: EC forecasts should be verified if possible, so Heidke with EC is used. Long range Heidke scores without EC can be an option in the future and would require some changes to ectype in the settings class (see EcType logic section below). EC forecasts are assumed to verify ⅓ of the time for Heidke. The scores for the below, near, and above categories are identical for both methods, only the scores for the total category may be different if there are EC forecasts.

The equation for heidke without EC is:

$Heidke_{noEC}$*=(numCorrect - numExpected)/(count - numExpected)*

where numExpected is count/number of categories and count is the number of valid fcst-ob pairs.

The equation for heidke with EC is:

$Heidke_{withEC}$*=((numCorrect of nonEC fcsts + numCorrect of EC fcsts) - numExpected)/(count - numExpected)*

where numCorrect of EC fcsts is (num of EC fcsts/numCats) or 1/3 of all EC fcsts when numCats is 3, and numExpected is (count/number of categories) and count is sum of valid EC and non EC fcst-ob pairs. HeidkeWithEC simplifies to HeidkeNoEC * coverage where coverage is number of non EC fcsts/count.

The Heidke score utilizes the number of correct and incorrect category hits. The values range from -50 to 100. A score of 100 indicates a perfect forecast, and a score of -50 indicates a perfectly incorrect forecast. Scores greater than 0 indicate improvement compared to a random forecast and indicate skill.

Heidke Dry Location Correction

The dry location correction is typically done for short range precip. The cases for which to do the correction are specified in Settings.getDryLocationCorrection. Currently the only situation that dry location correction is utilized is for **precipitation, Heidke score, lead time unit of "d", and spatial type (of input data) is station or gridded.**

Settings.getDryLocation returns true or false and is used by the Data class to load the dry percent climatology data if it is needed. The dry percent climatology data used by the Data class is the percentage of dry average windows (5 or 7 day means for extended range) per location for each valid period (5 or 7 day mean). In short, the dry location correction modifies the number of expected correct forecasts based on whether the location is arid, semi-arid, or normal (determined from the percent of time that location is dry based on climatology), and then collapses the 3 category forecasts to 2 categories (above or below median) for arid and semi-arid locations. To collapse to this 2 category system, forecasts, observations, and the expected number of dates expected to classify as near-median are changed to below-median for arid and semi-arid locations. For stations, the percent dry is based on stations with zero precip over the averaging period. For grids, the percent dry is based on grids with less than 1 mm of precip because it is very rare to get a completely dry grid due to spatial interpolation. See verification ticket #1278 for more details on gridded dry climatology.

The dry location correction using the methodology by Scott, Huug, Dave Unger is detailed here:
https://cpc-devtools.ncep.noaa.gov/trac/projects/Verif_System/wiki/CodingTeam/DryLocationCorrection

The possible range of precipitation Heidke Skill Scores using the dry station methodology described above theoretically ranges from negative infinity to 100. Note that scores less than -100 are rarely experienced.

RPSS

This method takes into account all categories at once so there are only scores for all categories together, and the scores for the separate categories for RPSS are all NaNs. EC forecasts are always included with this method since it does not use forecast categories directly, only the forecast probabilities. Ranked Probability Score (RPS) is a squared error score with respect to the cumulative probabilities for multi-category forecasts and whether or not the event occurred. The Ranked Probability Skill Score (RPSS) measures the improvement of the multi-category forecast relative to a reference forecast (the sample climatology).

The equation for RPSS is:

RPSS = 1 - RPS/RPS$_{\text{reference}}$ where

$$<RPS> = \frac{1}{n}\sum_{k=0}^{n}\left[\left(probB_k - obsB_k\right)^2 + \left(probN_k - obsN_k\right)^2 + \left(probA_k - obsA_k\right)^2\right]$$

where n is the total number of forecast-observation pairs, k is the index of forecast-observation pair, *B, N,* and *A* represent the below-,normal-, and above-normal/median categories, *prob* is the forecast probability of the specified category (expressed as a fraction), and *obs* is either 1 if the specified category verifies or 0 if it does not verify. The fractional probabilities of *B, N,* and *A* are cumulative and must add up to 1. For example, for a forecast of 50% (50%B,33%N,17%A), the cumulative probabilities are: B=0.5, N=0.833, A=1.0.

RPS$_{reference}$ *= 1/n * (probBref-obsB)*(probBref-obsB) + (probNref-obsN)*(probNref-obsN) + (probAref-obsA)*(probAref-obsA)*

where the values for probBref, probNref, and probAref are constants and are described in the constants section below. RPSS values range from -Infinity to 1. The RPS penalizes forecasts less severely when their probabilities are close to the outcome and more severely when their probabilities are further from the outcome.

Brier

The Brier score by definition verifies only for the forecast category. Since EC is not a forecast for a specific category, there should not be a Brier score for EC forecasts. The scores for all categories together are a weighted average of the B,N,A categories. The weights for each category are the number of valid forecast-observation pairs used to calculate each category's score. EC forecasts (zeros) should be removed from the forecast arrays before using this method. The Brier Score (BS) is a squared error score with respect to the forecast probability of the forecast category and whether or not the category occurred. The Brier Skill Score (BSS) measures the improvement of the forecast relative to a reference forecast (the sample climatology).

The equation for Brier skill score for a forecast category is:

$BSS = 1 - BS/BS_{reference}$ where

$$<BS> = \frac{1}{n}\sum_{k=0}^{n}\left[(prob_k - obs_k)^2\right]$$

where n is the number of locations.

where $BS_{reference}$ is:

$$<BSreference> = \frac{1}{n}\sum_{k=0}^{n}\left[(climProb_k - obs_k)^2\right]$$

If the category verifies, obs is 1, if it does not verify, obs is 0. The value for climProb is a constant and is described in the constants section below. BSS values range from -Infinity to 1. The BS penalizes forecasts less severely when the probabilities are close to the outcome and more severely when the probabilities are further from the outcome.

Reliability

There are two versions of calcReliability. For extended range forecasts, calcReliabilityNoEc is used. EC forecasts should be removed from the forecast array before the noEC version is used. For monthly and seasonal forecasts, calcReliabilityWithEc is used. The reliability is calculated for each category regardless of what category was forecast. Reliability scores for each category for each bin are always assessed separately to calculate the scores. The reliability for all categories together for each probability bin is calculated by adding up the reliability scores for each category for each bin (mathematically the same as a weighted average, where the weights for each category are the number of forecasts for each category). The scores for the below and above categories are identical for both of the calcReliability methods, but the scores for the N category and total category may be different if there are EC forecasts. See the next paragraph on how EC is handled. The x values are the average forecast probability for that bin. A separate method in StatsLibrary, calcAverageProbability, is used to calculate the average probability for each model and each bin to plot the scores at the correct x values. The reliability diagram indicates how the forecast probabilities match the percentage of time that the category actually occurred. The closer the plotted values are to the reference line, the more reliable the forecast. If the reliability is plotted above the reference line, the forecasts were overconfident (probabilities were too high). If the reliability is plotted below the reference line, the forecasts were under-confident (probabilities were too low).

The lowest probability for a manual forecast category of above or below normal drawn on the map is .3334. As a result that value is the lower threshold value for the bin ranging from .3334 - .4, and EC forecasts and probabilities of .3333 are in a separate bin. For reliability scores including EC forecasts, there is a separate bin at .3333 for EC forecasts (only probabilities of .3333 are included in this bin). EC forecasts are scored as forecasts for each category. The N category for this bin contains reliability scores for EC forecasts and forecasts where N is .3333. The A and B categories for this bin contain EC forecasts and also may contain forecasts of .3333 for their respective categories, although forecasts of .3333 for A or B only occur in rare cases when there is a forecast of .3334 for N. A sum (weighted average) of the reliability of the EC and N forecasts is used to calculate the reliability for the N/EC category for the .3333 probability bin. In short, the reliability for the .3333 bin can be attributed all or mostly to EC forecasts for the A and B categories and approximately even amounts of N and EC forecasts for the N category. The reliability of all categories together for this bin is the sum of the B, N/EC, and A reliabilities.

For each category and for each probability bin, the following equation is calculated:

$\text{reliability}_{\text{category}} = \text{\# obs}_{\text{category}} / \text{\# fcst}_{\text{category}}$ where category is either below, near, or above.

Reliability for each bin for all categories together is:

reliability = (# obs A / # fcst A) + (# obs B / # fcst B) + (# obs N / # fcst N)

where the # fcst of a category for a probability bin are obtained by counting the number of occurrences when there is a forecast for that category with a probability that falls within the probability bin. The # obs of a category for a probability bin are obtained by counting the number of occurrences where the forecast within that probability bin had that category.

Equal Chances Forecast Category (EC) Score Handling

A variable called *ectype* is set by the settings class so the stats class QC knows whether to keep or remove the EC forecasts and calls the correct version of the heidke and reliability methods (if Heidke or reliability is the score type). The options are: “withEC and “noEC”. For more info see ‘*Equal Chances Forecast Category (EC) Handling*’

Because the inclusion of EC forecasts can change the resulting scores, there are variations of the heidke and reliability methods that calculate with and without EC in StatsLibrary.java. There is only one RPSS calculation method (calcRpss) and it always uses “withEC” in its method. Currently, the software automatically figures out whether to include EC (“withEC”) or exclude EC (“noEC”) based on some of the settings.

Constants

The number of categories is set to 3 for calcHeidke, but the code would probably still work if this was changed to a different number. This has not been tested. calcRPSS, calcBrier, and calcReliability are set up to verify for 3 categories, but this is not set by a variable. These methods are set up to use the forecast probabilities by directly accessing the index (0,1,or 2) of the forecast probabilities, so in order to verify a different number of categories some modifications to the code would be needed. For the reliability diagram, the number of probability bins and the probability bin thresholds are set in the Stats class and passed to calcReliability. See the constants section under Stats for more information. RPSS climatological reference probabilities (probBref, probNref, probAref) are set to .3333 for Below, .6666 for Near Normal, and .9999 for Above. This is to match the values in Scott's verification. 1/3 for Below, 2/3 for Near Normal, and 1 for Above would also be appropriate values. For Brier, the climatological reference value (climProb) is 1/3. The scores are rounded to 2 decimal places by calling FormatLibrary.roundToDecimal(scoreArray,2) in the Stats class.

MathLibrary.java

Contains simple math functions over arrays (things not part of the basic java library). For example, determining the highest, lowest, and average score for a given data set.

Score Summary

There is a score summary which contains information such as the minimum/maximum score values and reference point of occurrence for each model and each category (total, below, normal, above). This information is contained in a 3-D array which can be retrieved by calling StatsLibrary.getScoreSummaryArrayEachModel() and passing the stats (ie. by resultsObj.getStats()) and resultsObj.getReferenceArray()) and reference arrays to the method. It is left up to the initiation classes (the applets) to perform this function after the necessary stats and reference data is loaded into the results object. There is no score summary for reliability. Below is a description of the dimensions and data that is contained:

1st dimension : models (forecast source), index matches order selected in the settings.
2nd dimension : category (Total, B, N, A) where B is below-normal, N is normal, and A is above-normal. The order listed of the categories corresponds to the 0-3 index of the array dimension.
3rd dimension :

index #	information
0	max value
1	date(s) or location(s) of max value. If more than one value, comma separated.
2	min value
3	date(s) or location(s) of min value. If more than one value, comma separated.
4	average value
5	# of values making up average
6	# of values making up the max value
7	# of values making up the min value.

Appendix E : Format Package

The format package format classes for back-end handling of formatting for various parts of the verification tool. This includes methods and classes for generic functions for formatting as well as reading and writing XML. If the output type is “**ascii**” and this method is called, an error is returned since the creation of an ASCII file should be called from the WriteLibrary.class. The StaticRunDriver.class driver should be calling the writeToAscii() method in WriteLibrary.class which does the formatting and creation of the output in the same method. The below sections will outline how formatting for the different types of output works. This section explains how the formatting for charts and maps works separately than the ASCII files.

Below is an overview of the classes in the Format package and a brief description of their purposes:

- Format - Formats a loaded Stats object with data and can return formatted output based on the type of plot or output format requested.
- FormatLibrary - Contains methods that the Format class accesses that performs the formatting functions. Also contains methods to do more general conversions, ie, converting between primitive types and obtaining alternate text versions of settings options.
- FormatHTMLLibrary - Formats HTML used in the web application version, such as the summary information displayed in the bottom panel of the web application verification page.
- XMLCreator - Method that uses level-2 DOM methods to create XML formatted statistics output. This can currently create formatted output for a JClass Chart.
- Chart - Constructs a chart object with specific attributes that the XMLCreator uses.
- GoogleEarth - Constructs an object used to create the attributes used for the Google Map display.
- IconMap - Formats data for legends for the Google Map plotted results.
- XMLCounter - Counts the number of times a specific XML element is encountered.
- XMLReader - Reads and parses an XML file, specifically a settings XML file that contains the settings parameters to run the verification software tool for. This utilizes method(s) in XMLCounter to obtain the number of settings groupings to initialize arrays.
- Table - Formats for table output (not created yet).

Formatting for charts and maps

Charts and maps are created in the verification software by providing XML formatted data to the display package. XML formatted data is passed to the PlotChart.class that utilizes JClass software to create a chart. KML formatted data is passed to Javascript that pushes the data to the GoogleMaps server to create a map.

To create the XML for either the chart or map, first a Format object is instantiated (typically done in the VerificationDriver.class). To create the object, you pass the loaded Stats, Settings, and Data object. Then the formatData() method (associated with the Format object) is called to perform the formatting functionality. The formatData() method in Format.class contains if statements for each of the settings output types - currently “**chart**” and “**map**”. These if statements call the appropriate method for formatting data for those output types in the FormatLibrary.class. FormatLibrary.class contains methods such as:

- formatForJClassTimeseries() - formats for a JClass chart
- formatForJClassReliabilityDiagram() - formats for a JClass reliaibility diagram chart
- formatForGoogleEarth() - formats for a Google Map on the web application version of the VWT

Each type of output that needs a different format of XML is represented by a different class that creates an object based on the output type (Chart.class and GoogleEarth.class objects). These format objects contain the data that would be used to build the XML formatted String. Each of the format methods listed above create the format objects and load the data to format into the objects by extracting information from the settings, stats, and data objects that the above methods receive. Then these methods pass the format objects to the appropriate XML creator method that produces the final XML formatted String of data.

Once the XML is created, various levels of formatting classes pass the XML String further up until the applets get the updated XML which is then passed to the output creation packages. The details on how the XML creation works is in the howToWriteXML.doc in the /docs directory of the verification software.

The XML creation class utilizes W3C Level 2 Document Object Model (DOM). These classes are already included in the software and do not have to be downloaded separately.

Formatting for charts

The chart uses the same x values (dates) for each set of y values (the scores). The interactive JClass GUI options (pop-up box after right-clicking on the web application JClass chart) allows users to select a different type of chart for the data to be displayed on so that the data can be displayed on other types of charts. Below is a high-level process flow diagram of the formatting process for JClass line plots. Below is a high-level diagram of the process flow of how formatting for JClass line charts works in the verification tool. The diagram is for the chart applet initiating the process, but the command-line run version can also initiate and produce a chart. The StaticRunDriver would replace the ChartApplet.

For a developer to add a new XML formatting type, they would have to perform the following steps:

1. Create a class representing an object for the new format type
2. Edit XMLCreator.java - adding a new method that performs the creation of the new XML format with the XML tag attributes and values associated with them.
3. Add a new method in FormatLibrary.java that accepts objects with the data used to insert in the XML and instantiates an XMLCreator object and calls the appropriate formatting method.
4. Add an “else if” statement in Format.java that calls the appropriate FormatLibrary.java

format method depending on the outputType (or another settings type).

5. Use appropriate get methods/ensure get methods will properly pass your formatted XML back to the initial code that needs it.

As mentioned above, the XMLCreator.java code that you would add to create the XML uses DOM2. Please refer to the howToWriteXML.doc file in the software /docs directory. As a reference, the XML schema for JClass charts is in the /docs sub-directory of the VWT software (JClassChart.dtd).

Formatting for reliability diagram charts

The process flow for formatting the results into XML is almost the same as the process flow of chart formatting. The reliability diagram uses unique x values (forecast probabilities) for each set of y values (observed probabilities). The reliability diagram requires a few extra steps in to set up the x axis values. For example, xml for the the diagonal reference line is created first using a reference array which contains the probability bin interval labels. Then sets of x axis values are created for each dataset. The *formatForJClassReliabilityDiagram()* method in *FormatLibrary.java*, gets the reference array form the stats object, which is pre-set in the object and does not change based on the scores. The reference array is used to plot the axis labels and the diagonal reference line directly on those labeled tick marks. For the reliability diagram a separate Chart.java format object is used and a different method (with reliability in the title) in *XMLCreator.java* is used to format the data into XML.

Formatting for Google Earth

The formatting used to create the Google Map plot uses KML, a form of XML, specifically used for geographical plots. The method of creating the Google Map plots vs. the chart differs most in how the KML is passed to create the graphic which is discussed in the display package documentation (Appendix F : Display Package). The actual formatting process flow is almost identical to the chart formatting process flow except for the specific XML attributes and details of the methods used for formatting. Code for functions related to spatial plotting using Google Earth, including the dots for scores and title, is in /web/library/GoogleEarth.js. Below is the process flow diagram for the Google Maps. The convention for using the term Google Earth in the codes and scripts is because it is the general API that is used for Google maps.

Formatting for ASCII files

The creation of ASCII files with data involves the formatting and creation of the files by calling one method, rather than using the formatting and display package for the formatting and creation of output like the chart and map creation. The driver, StaticRunDriver.java, is responsible for calling the method that formats and creates the output after the main driver (VerificationDriver.java) method runDriver() is ran.

The steps performed in the software for creating the ASCII files are:

1. The global environment variable ${VERIF_HOME} must be set prior to running this since this determines where the root of the /output directory is. This should have been

done as part of the installation and building step in the .cshrc file on the machine where the software is installed and running.

2. The StaticRunDriver.java needs to call the method to set this global environment variable in the driver object before runDriver() is called. Once the VerificationDriver.java object is created (ie. driverObj), the method setHome() should be called, passing a String representing the root directory of the verification software. The main() method in the StaticRunDriver.java sets the “home” variable by automatically looking for the value on the machine.
3. Running the driver so that the settings, results object are properly loaded
4. Get ASCII file name which is built using various pieces of information from the loaded objects (created by *FormatLibrary.getImageFileName()* or *FormatLibrary.getAsciiFileName()*.
5. Call the writeToAscii() method in WriteLibrary (static method, does not require a WriteLibrary object to call the method), passing the stats and results object, and file name to the method.
6. The writeToAscii() method formats the data by appending difference pieces of the results and information to a String in memory.
7. The string with the data is written to a file with the passed file name in the /output sub-directory using Java’s PrintWriter functionality.

Appendix F: Display Package

Creating chart displays

The charts are created by using JClass DesktopViews software developed by Quest Software : http://www.quest.com/jclass-desktopviews/ . The verification software uses the methods and classes in the JClass software to create charts on the web application or command-line run version of the tool. Since the JClass software is written in Java, certain parts of the chart can be modified by using Java configurations. Therefore, two main ways that the charts are created and displayed are from 1) utilizing the JClass methods and properties 2) utilizing inherent Java methods and classes to modify and set properties of chart components. Viewing the top part of the PlotChart.java source code gives the developer an idea all the various packages of JClass and Java that are used to create the chart display.

Overall chart creation and update steps

1. makeChart() is initiated when the ChartApplet.java init() method is ran.This method creates the chart and sets generic chart properties that would apply to the chart across all runs of the chart updates.
2. The applet creates a new PlotChart object, which sets the layout and does generic setting.
3. Data for the chart is updated, by passing an XML string. The XML schema is based on

the JClass schema. Some axis and label properties are retrieved by get methods to be modified in method updateLabels() later on.

4. Labels on the chart are updated as well as other chart properties:
    a. x-axis and y-axis labels
    b. title label
    c. markers are updated

See the “JClass_Desktop_Tutorial.pptx” powerpoint presentation as a good intro for starting to learn and understand how the JClass software works that creates the charts.

Below is a process flow diagram of the display package of the web application version of the VWT..

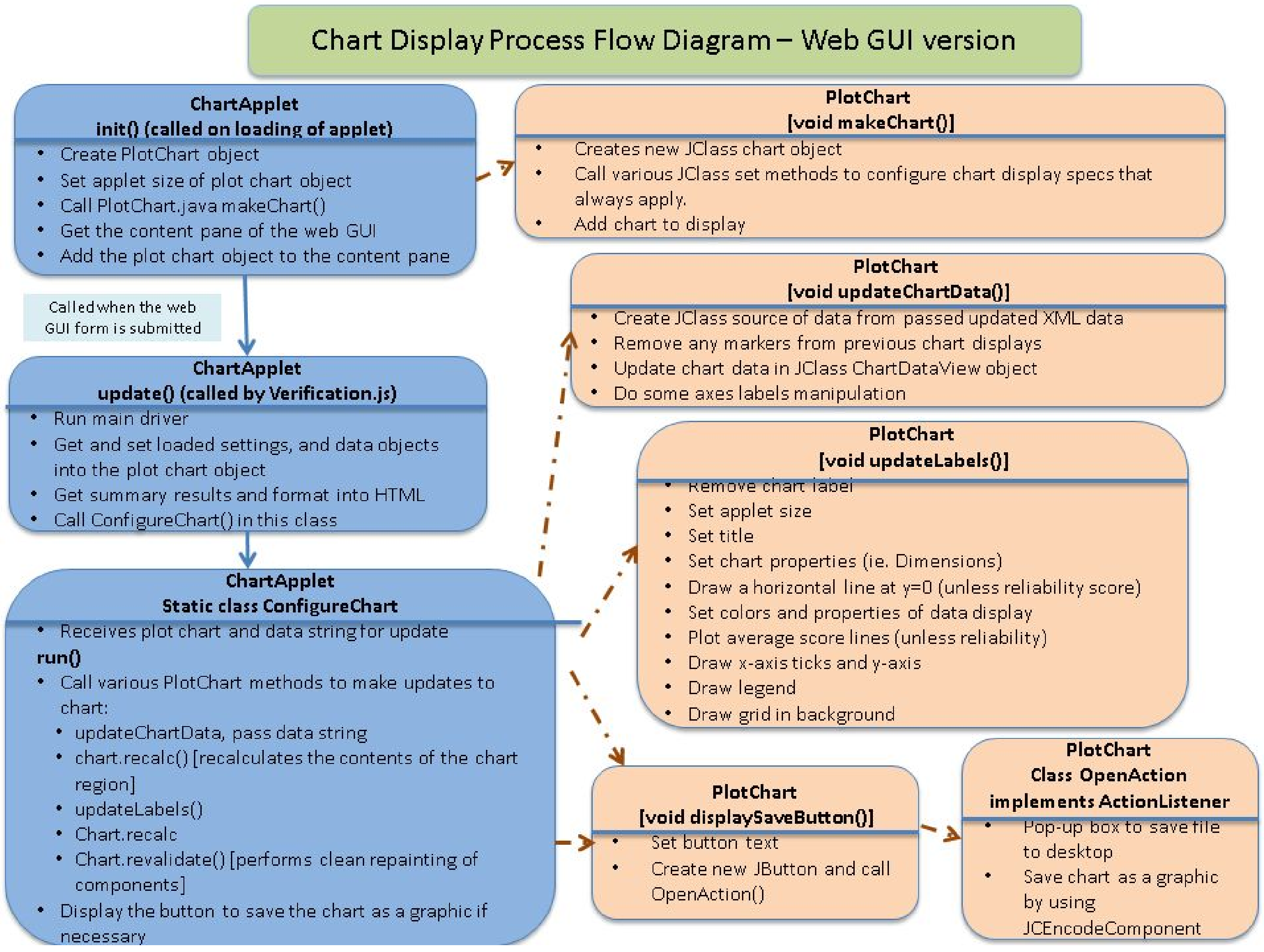


Chart Display Process Flow Diagram – Web GUI version
ChartApplet
init() (called on loading of applet)
Create PlotChart object
Set applet size of plot chart object
Call PlotChart.java makeChart()
Get the content pane of the web GUI
Add the plot chart object to the content pane
Called when the web GUI form is submitted
ChartApplet
update() (called by Verification.js)
Run main driver
Get and set loaded settings, and data objects into the plot chart object
Get summary results and format into HTML
Call ConfigureChart() in this class
ChartApplet
Static class ConfigureChart
Receives plot chart and data string for update
run()
Call various PlotChart methods to make updates to chart:
updateChartData, pass data string
chart.recalc() [recalculates the contents of the chart region]
updateLabels()
Chart.recalc
Chart.revalidate() [performs clean repainting of components]
Display the button to save the chart as a graphic if necessary
PlotChart
[void makeChart()]
Creates new JClass chart object
Call various JClass set methods to configure chart display specs that always apply.
Add chart to display
PlotChart
[void updateChartData()]
Create JClass source of data from passed updated XML data
Remove any markers from previous chart displays
Update chart data in JClass ChartDataView object
Do some axes labels manipulation
PlotChart
[void updateLabels()]
Remove chart label
Set applet size
Set title
Set chart properties (ie. Dimensions)
Draw a horizontal line at y=0 (unless reliability score)
Set colors and properties of data display
Plot average score lines (unless reliability)
Draw x-axis ticks and y-axis
Draw legend
Draw grid in background
PlotChart
[void displaySaveButton()]
Set button text
Create new JButton and call OpenAction()
PlotChart
Class OpenAction implements ActionListener
Pop-up box to save file to desktop
Save chart as a graphic by using JCEncodeComponent

Fig. 9: High-level process flow diagram of how the display package works to create a chart in the web application version of the VWT.

Below is a process flow diagram of the display package of the command-line run version of the VWT:

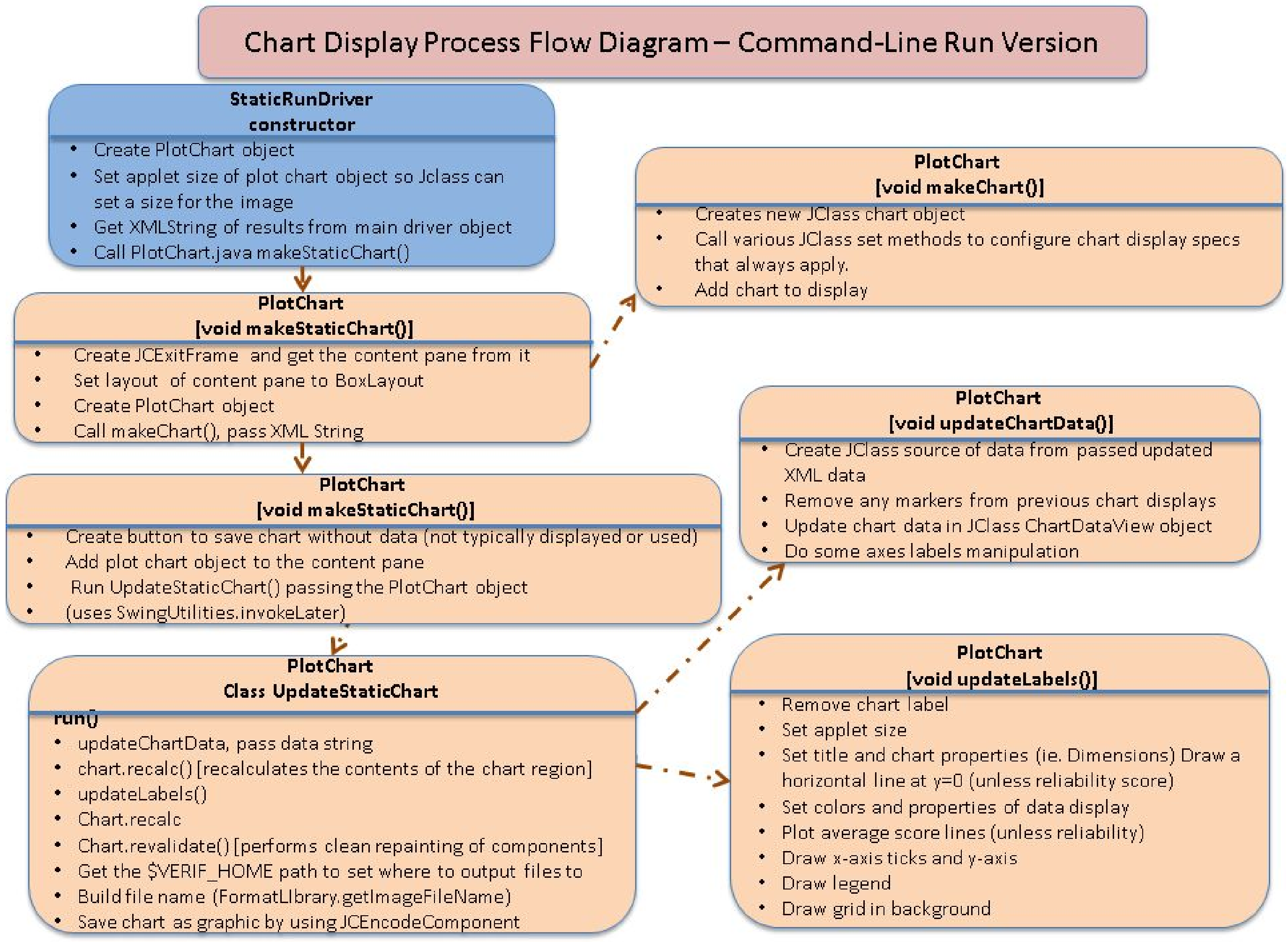


Chart Display Process Flow Diagram – Command-Line Run Version

StaticRunDriver constructor
- Create PlotChart object
- Set applet size of plot chart object so Jclass can set a size for the image
- Get XMLString of results from main driver object
- Call PlotChart.java makeStaticChart()

PlotChart [void makeStaticChart()]
- Create JCExitFrame and get the content pane from it
- Set layout of content pane to BoxLayout
- Create PlotChart object
- Call makeChart(), pass XML String

PlotChart [void makeChart()]
- Creates new JClass chart object
- Call various JClass set methods to configure chart display specs that always apply.
- Add chart to display

PlotChart [void makeStaticChart()]
- Create button to save chart without data (not typically displayed or used)
- Add plot chart object to the content pane
- Run UpdateStaticChart() passing the PlotChart object
- (uses SwingUtilities.invokeLater)

PlotChart Class UpdateStaticChart

run()
- updateChartData, pass data string
- chart.recalc() [recalculates the contents of the chart region]
- updateLabels()
- Chart.recalc
- Chart.revalidate() [performs clean repainting of components]
- Get the $VERIF_HOME path to set where to output files to
- Build file name (FormatLIbrary.getImageFileName)
- Save chart as graphic by using JCEncodeComponent

PlotChart [void updateChartData()]
- Create JClass source of data from passed updated XML data
- Remove any markers from previous chart displays
- Update chart data in JClass ChartDataView object
- Do some axes labels manipulation

PlotChart [void updateLabels()]
- Remove chart label
- Set applet size
- Set title and chart properties (ie. Dimensions) Draw a horizontal line at y=0 (unless reliability score)
- Set colors and properties of data display
- Plot average score lines (unless reliability)
- Draw x-axis ticks and y-axis
- Draw legend
- Draw grid in background

Fig. 10: High-level process flow diagram of how the display package works to create a chart in the command-line run version of the VWT.

Understanding JClass charts

The basis of JClass software is that there is a chart object, specifically of type JCChart. From now on, I will refer to a JCChart object as simplay a chart object. This should not be confused with the chart object from the format class (format/Chart.java) which simply contains attributes of the chart that is used for XML formatting.

Manipulating components of the chart

The JCChart chart object inherently contains other objects that represent components of the chart, such as the axes, and labels. By creating a chart object, these other objects get created as part of the chart object. By understanding how to retrieve some of these other objects, as well as create or modify them, developers can configure the chart look and feel and functionality. Since many pieces of the chart are really Java Swing/AWT GUI objects at the core, you can often manipulate and configure chart components beyond what JClass methods offer. For example, JClass has a method that allows the border color and width to be set. The border property (BorderFactory) is actually a Javax Swing class that can be set. Below is a sample line of code that shows this:

```
legend.setBorder(BorderFactory.createLineBorder(Color.black,1));
```

Often to modify a component of the chart, you have to retrieve that component, typically by calling a specific get method, and then applying a set method to edit that property. This ends up looking a little like a chain of functions strung together in one line. Below is a sample line of code that shows this:
The below performs necessary steps to set the style of the line used in the grid of the chart.

```
JCAxis xAxis = chartObj.getDataView(0).getXAxis();
xaxis.getGridStyle().setLinePattern(JCLineStyle.SHORT_DASH);
```

Developing with JClass

The goal of this section is to enable developers to understand how JClass in general works, share important coding tips, and teach developers how to edit or add to chart functionality. The section below “Understanding chart properties and features” explains more details of specific VWT chart settings and functions. This will also outline useful resources developers can use to develop JClass charts.

You should read the JClass Desktop manual (location listed below in “Resources”) to understand the hierarchy of the components and objects and what you can do with JClass. This will give you a good idea of the overall framework of how JClass produces charts. Some tricks and workflow ideas will be provided in “Understanding chart properties and features” below . There is also a Powerpoint presentation that the VWT developers created JClass_Desktop_Tutorial.pptx, which is located in the VWT /docs sub-directory. An example of how to modify chart properties is included in the presentation.

* Important note: The documentation that comes with the JClass software (either the jcchart.pdf or docs API) are sometimes outdated or inconsistent between each other. An example was that one of the method usages were inconsistent between the jcchart.pdf and API. A support request was made to Quest Software regarding this. However, response by Quest may depend on whether the person(s) issuing the request currently has a valid support license agreement. This page contains updated release notes and bugs as well as documentation : http://www.quest.com/jclass/readme-desktopviews.html#docs

Things You need to know to modify JClass charts
Before you tackle learning JClass, there are essential Java and programming things you need to know. If you do not know what any of the following items mean, you need to research (google, online tutorials, books) them!

- How to use the JClass and Java doc API (ie. to understand how to modify Java Swing/AWT components)
- What Java Swing GUI is and how components of them work. JClass utilizes many Swing pieces! You will need to distinguish what you can manipulate using Swing vs. the JClass specific classes. Also, most of these graphical components in JClass are built out of Swing components! Such as panels, shapes, etc. that JClass has created special designed wrappers or classes for.
- Java packages, how to reference them, how to import them.
- The components of charts and what the objects are called (JCAxis, JCChart, etc.)

The main items you must do to use JClass in your Java code :

- Import the appropriate packages and associated classes. (ie. import com.klg.jclass.chart.*)
- Some JClass classes use Swing components. Often you can manipulate these. Import these components as well.
- Create an object if necessary to use the associated JClass functionality. This often involves creating a new object but really retrieving an object that already exists on creation of the chart object. Then you can often use set methods to modify the specific retrieved component of the chart.
- Find the method(s) that you would like to use methods.
- Call these methods passing arguments that you want. For example there are a lot of set methods in JClass to set attributes of the chart object. You must understand the hierarchy of these JClass objects and attributes by reading the documentation carefully to code with JClass. There is some other tricks to finding out the hierarchy of certain components. See the “Understanding chart properties and features” section.

Resources

- JClass pdf of documentation, available in the verification /docs directory (jcchart.pdf) as well as the docs directory of where the JCLass software is installed ($ROOT/JClassDesktopViews/docs/chart/ directory in a browser, where $ROOT is the path of the directory above where JClassDesktopViews software is installed).
- JClass Javadoc API - comes with the JClass software, navigate to the $ROOT/JClassDesktopViews/docs/chart/ directory in a browser, where $ROOT is the path of the directory above where JClassDesktopViews software is installed.
- Online JClass readme page: http://www.quest.com/jclass/readme-desktopviews.html#docs
- Powerpoint tutorial created by the verification developers as an intro to JClass and using it: JClass_Desktop_Tutorial.pptx inside the VWT /docs subdirector.

Understanding chart properties and features

This section outlines how the chart properties are configured. There were some solutions provided by Qwest to solve some of the issues or functions that we wanted. A list of features and set properties used in the chart of the VWT specifically, are listed below and then an explanation of how the code does them is provided.

- To load chart data :

- Create ChartDataModel object and adding a new JCStringDataSource. In the VWT an XML String with formatted data is passed.
- Updating chart data
    - Web application Version - The chart update class UpdateChart (in PlotChart.java) uses event dispatcher methods/applications, including Applet usage of PlotChart. The web application version of the verification web tool accesses this method to update the chart dynamically. This optimizes updating of the chart without recreating the entire chart object each time selections are changed by the user in the web tool. This method calls other methods to recalculate and redisplay parts of the chart with updated information and data.
    - Command-line run version - The chart update class UpdateStaticChart (in PlotChart.java) updates in a similar way to the web application version. In the command-line run version, however, the graphic is automatically saved as a graphic file to the process /output sub-directory.
- Chart layout settings
    - BoxLayout
    - Set background color (white)
- Enabled JClass Chart Customizer (pop-up box with user changeable chart options) - chartObj.setAllowUserChanges(true). Also set the trigger for the GUI to display.
- Anti-Aliasing turned on for plot smoothing - chartObj.setAntiAliasing()
- Legend labels lined up- By default the legend labels are just listed one after another and JClass figures out where to start the new row according to the available space after the chart is displayed. In the VWT, since we often have a line drawn of the average score associated with each line representing the scores, we wanted the legend to show the average and scores of each forecast source in each column in the legend. This was achieved by using JCMultiColLegend(). Set the number of columns and rows (so that the width is consistent of the legend box). The number of columns should be the same number as the maximum number of allowed selected forecast sources for one run-time.
- Other legend settings
    - set position relative to chart- Used setPlacement relative to the x-axis, used setLayoutHints()
    - Made legend horizontal (setOrientation)
    - Anchored the corner (setAnchor)
    - Set border style using BorderFactory (setBorder)
    - Set legend to visible (setVisible(true))
    - Set font style - Created a Java Font object, created a JComponent called legendFeature, which was retrieved from the JCChart object (chartObj.getLegend()). Then, was able to set the font for the legend.
    - To display the legend, the code is : codeObj.getLegend().setVisible(true)
- Axis settings

To manipulate the axes, because the axes to manipulate are associated with JCChart's ChartDateView object, to manipulate the axis, you must retrieve the x-axis and y-axis associated with the dataView and create this as JCAxis objects:
ChartDataView dataView = chart.getDataView(0); // The 0 just means the chart
- Create the xaxis and yaxis by retrieving it

```
JCAxis yaxis = dataView.getYAxis(); // do same for xaxis
- now you can use JCAxis methods
xaxis.setAnnotationMethod(JCAxis.VALUE_LABELS);
```

- Markers - These are markings, such as lines that are plotted additional to the lines representing the score data values.
    - Old markers are removed before displaying new ones- To do this:
      1) Create a ChartDataView object by retrieving the dataview from the chart object. Use an Iterator to remove all found markers in the ChartDataView object.
- Labels - These are the labels associated with values along the x-axis and y-axis.There is some manipulation in the code that changes point labels to value labels so that they can be queried later in the updateLabels() method in PlotChart.java. The method updateLabels() performs the following:
    - label values for x-axis, y-axis, and the title are updated
- Line styles associated with data plotted:
    - Since overlapping of labels sometimes makes it difficult to see multiple dots representing values, different sizes of the symbols representing the score values are used for each of the forecast sources displayed. The size of the forecast source symbols start at the largest set value and loop decreasing in size by 1 until the minimum set symbol size. If there are more forecast sources than unique symbol sizes, the minimum set symbol size is used for the rest of the forecast sources' symbols. Variables minSymbolSize and maxSymbolSize are responsible for the size bounds in the PlotChart.java updateLabels() method.
- A horizontal line at y=0 is drawn on the chart except for the reliability diagram- This is useful to indicate a threshold where the skill score is zero. A JCMarker is created (variable markerZeroValue) and then utilizes get and set methods to create and display the marker.
- Colors and style of data symbols and lines and associated markers reflect a 3-category forecast color scheme - The colors of lines and markers are set by order that the categories go by. In the situation where scores for each category are displayed, the order of categories are below normal, normal, and above normal categories and correspond to cool, neutral, and warm colors for symbols, lines, and markers of each forecast source. The order of set colors are the same, regardless of whether separate categories or a combined category score is plotted. Therefore, if a combined category score is displayed, the colors used just don't reflect any extra information. If there are many forecast sources gray colors are used since only 11 colors are set in the code. If the number of colors are not sufficient enough for the number of lines drawn, then a warning error should be displayed.

  The symbol shapes are also cycled through to make each line with data more distinguishable as well.
- Lines representing the average skill score for each forecast source is plotted if the category type is "total" (combined category scores) and is not the reliability score
    - The legend rows and columns are calculated to properly have the data line and

associated average line of a specific forecast source be in the same column.
    - An array of markers for each of the average lines needed (of type JCMarker) is created and is the size of the number of forecast sources represented. In JClass, each forecast source's data is referred to as "series". These series are retrieved from the ChartDataView object. A for loop is used to create a JCMarker marker set to the average score value as the y-axis value to draw the marker (average line) for each of the series (forecast sources with score results). JCMarker get and set methods are used to configure the styles of these lines, then the markers are added to the data view for display.
- Ticks are only drawn on the x-axis where JClass decides to display the labels- These labels on the x-axis could be dates for example. JClass automatically figures out the interval to display x-axis labels at so that text is not overlapped. So, even though JClass always draws a point on the chart representing every data value passed as the data string, JClass figures out the interval to put the labels on the x-axis, skipping the same number of reference data values that would be on the x-axis each time a label is drawn. However! by default, the ticks that JClass draws is not drawn just with the displayed labels. Rather, tons of ticks are drawn between each label causing an illegible x-axis display sometimes. This problem was brought up to Quest Software and Milton Villegas solved this (Case ID: 801933 submitted by Melissa Ou from NOAA, specifically from the Climate Prediction Center).

The solution involved using value labels, since there was no workaround with point labels in JClass. The logic in PlotChart.java involves retrieving the labels that JClass generated. For all of the labels on the x-axis, if the label in the chart is not visible, then the ticks are not drawn for that label point (JCValueLabel method setDrawTick(false)). Additionally, the grid lines drawn on the chart are also not drawn at label points on the x-axis where there is no label drawn.

- Special chart for reliability skill score diagrams- Since the reliability diagram has a different format than other charts, special settings are done in the chart code. The x-axis represents forecast probability, and the y-axis represents the observed frequency that the forecast probabilities were correct.
    - Labels for x-axis are retrieved from the stats object method getProbabilityBinAxisLabels()
    - Slightly different order of setting line color and styles for data display. In this case, a black reference line is plotted first before setting the rest of the colors and styles for the data lines.
    - An additional array containing the average forecast probability for each model and bin is used to plot each score dataset with its unique x axis values, ie the average forecast probability of all forecasts for that bin. (For a timeseries, all datasets share the same x axis values, ie. dates).

Tips and Notes About the Chart Display

- Since the VWT uses JClass in a web application where charts are created dynamically, markers are removed before being added after each time the settings options are submitted to the VWT. This precludes overlayingof markers each time the VWT runs. There is also a trick used so that the “Save chart as graphic” button does not display multiple times.

- There are often many ways to modify the chart in code. If you want to set many of the properties of a component (ie, the line style, color, size, etc.) or if the code creates and modifies the chart a certain way, you will often have to use more steps in achieving what you want, or use the methods in a way that works with the rest of the code.
- Anything drawn on a chart has a z-index, representing the order that the item is displayed. Depending on what you want to be displayed on top, you will have to use various methods to set this. For example, many JClass components have a method setDrawnBeforeData() which you pass a boolean value to to set whether you want the item drawn before data values. Since you often want the scores to be the easiest to read, and therefore on top of any other items on the chart, you would set other components to have the value “true” set for setDrawnBeforeData (so data is drawn last and on top).
- Sometimes you will need to use ‘recalc’ methods to make updates to the chart.

Common developer problems when creating and modifying chart properties

- You have used methods to create and add a chart component but it does not show up when you run the code!
  - JClass often has various methods that you need to enable or set to have the component display properly. This can be a boolean method, ie. setVisible(true), or a component-specific set method, ie. xaxis.setGridVisible(true). Without setting these other visible methods, often the components will not display. A good trick is to search in the JClass doc API for the word “visible” or “display” to see if there are any methods related to this for a certain component.
  - You may have not properly retrieved a component which will enable you to use set methods to set properties. Look at the documentation or get methods of a JClass component to see whether a related object can be retrieved and then set.
- Compile errors complaining that a method cannot be found- The method is there in the documentation and you have utilized the method correctly: so what is the problem? Often you may not realize that you have to import another package within JClass Chart, or even a Java Swing or AWT graphical package in order to modify or add a component. Look through the API documentation
- You are trying to set a style setting, such as color and the method asks for the color to be passed as the argument to a method, but error at compile time complaining that it can’t find the argument or incorrect type of variable passed- A good example is the fact that a JCChart method to set the background color is shown in the docs API as usage: void setBackground(java.awt.Color c)
  If you do chartObj.

How to change the size of the chart
To change the size of the chart that displays in the web application:
Edit the /web/include/chart.php. You can change the width in height in the lines:

```
 <object id="applet" type="application/x-java-applet" width="820" height="600">
```

To change the size of the chart in the command-line run version:
Edit the source code in StaticRunDriver.java in the driver package. The constructor contains the line that sets the width and height:
(The below may not reflect the current dimensions set, the exact dimensions may have been altered since this section was written).

```
// Set the Applet size so JClass can set a size for the image
chart.setAppletSize(820,600);
```

where the first argument passed is the width and the second is the length. These must be specified as type integer.

JClass Chart Customizer
By right-clicking over a JClass chart, you can open the JClass Chart Customizer. The software is currently set to enable this pop-up GUI that allows users to dynamically modify features and see chart properties.

Appendix G: Quality Control Documentation

There are two levels of quality control (QC) performed by the verification software for scores except for reliability and RPSS separate category scores, which have one level of quality control for reliability diagrams. This section describes the various methods of quality control. All QC is performed separately for each forecast source if multiple sources are in a run.

There are 2 settings in the configuration (.conf) file that is used to determine the thresholds for both levels of QC, and may or may not be used depending on the type of run. The variable names are goodDataThreshold (QC1), and goodScoreThreshold (QC2) for non-reliability scores and goodScoreThresholdReliability for reliability (unique combined QC). If these values are not set, an error will be thrown and the process will be killed. All quality control is initiated by the Stats class. There are methods in the qc package in QCLibrary.java that are used to perform these quality control (QC) functions.

The number and percentage of valid dates/locations/forecasts making up each score and the entire plot are printed in the ascii files for reference (there is some differences of printed information depending on the type of run). See the section on ASCII File Output section of the Command Line Usage section for detailed information.

More information and current status of QC is available in the Google Doc :

https://docs.google.com/a/noaa.gov/document/d/1kVTfRg-sxzSjTFlciB3elMSoCDHrgc2aC11vZ71vym8/edit#

There is also QC-ing performed by the upstream Perl scripts on the forecast and observed category and forecast probability data imported into the database is performed to make sure the data in the database is of the highest quality. The importation of data into the database is done by the database importation scripts.

The quality control done in the upstream import scripts include:

- Setting any missing data from ascii files (values of -99 or less) to NULL.
- Setting observations of ‘0’ to NULL.
- Converting percentages (0-100) to fractions (0-1).
- Fixing an issue with the manual probabilities resulting in negative probabilities.
- Forcing EC (monthly and seasonal only) forecasts to have probabilities of 0.3333.
- Ensuring probabilities are bound by 0 and 1.

Equal Chances (EC) forecast impact on QC

Stats.java calls a method *dataObj.getNumDataByCategory(),* as mentioned in ‘Stats usage of EC type’ which returns the number of data points in each category. This is mainly used for getting the number of EC forecasts. By default the variable array containing this is initiated with ‘0’ (assessed for each reference point of output dimension, for 4 categories - below, normal, above, equal chances). The values are only overwritten by actual values if ‘noEC’ is selected. The separate sections below covers in more detail how this information is utilized in the QC.

If the option for ‘noEC’ is selected, the number of forecasts with EC is counted and subtracted from the expected number of fcst-obs pairs to distinguish between values that are actually considered low-quality (too many NaN fcst/obs values) and forecasts that were replaced with a NaN value during run-time to remove values that contain a ‘0’ due to a part of the process that replaces forecast values with a value of ‘0.0’ (EC category) with a ‘NaN’ (removeZeros method) during ‘noEC’ runs. Without subtracting the number of forecasts that actually have a valid value but contained an EC category, the percent valid fcst-obs pairs would be incorrect (not reflect the actual amount of good data due to the run-time replacement of EC values with NaN) and the percents may often end up below the QC 1 threshold incorrectly. There are versions of QC equations below that account for it and are described.

Dry Station Correction impact on QC

QC-ing via removing of scores based on data (QC 1) or removal of all scores (QC 2) does not get impacted by dry station correction. There is a dry station correction version of the method that calculates the count and percent of good data (QCLibrary.countGoodDataPairsDryCorrection). However, the only difference between this method and the non dry correction one, countGoodDataPairs, is that this one calculates the % dry station data that is missing and prints it to the log file. The actual scores do not get evaluated based on this percentage at this time because it was discovered that the dry station data does not have as many stations as the forecasts or observations and it was unfair to penalize the data based on this.

Quality control of scores except reliability

There are 2 levels of QC except for reliability and RPSS separate categories (for Heidke and Brier):

QC 1 (assessment of data that creates scores) :
Evaluation of whether there is sufficient forecast and observation data to produce a score (ie. written to as the value in each row of the ASCII file). This is done for all score types except reliability and separate category RPSS. A forecast-observation (fcst-obs) pair at a reference point (determined by output dimension) - either each time step or location (for each forecast source) is considered “good” or “valid” if neither the forecast or observation data in this pair has a ‘NaN’ value. The forecasts used are in category format when this is assessed.

If the percentage of data making up a single score does not meet the goodDataThreshold, that score for the time or location is replaced with a NaN. For a timeseries (time as output dimension), this would be any date that does not have enough locations, and for the spatial output dimension, this would be any location that does not have enough dates.

In Stats.java there is a loop for each forecast and reference point that calculates the percent of good data for each of the reference points - QCLibrary.countGoodDataPairs or QCLibrary.countGoodDataPairsDryCorrection (see QC dry correction for more information). For non-reliability scores, this method is called and calculates the actual % of “good data” whereas reliability does the order of figuring this out slightly differently.

The equation to calculate this using “withEC” is (QCLibrary.countGoodDataPairs or countGoodDataPairsDryCorrection):
% good data (pairs) = # of non-NaN forecast-observation pairs / # of expected pairs * 100.

For “no EC” (QCLibrary.countGoodDataPairs or countGoodDataPairsDryCorrection)::
% good data (pairs) = # of non-NaN forecast-observation pairs / (# of expected pairs - # EC fcsts)* 100.

After the good % data is obtained for each reference point, QCLibrary.removeScoresWithLowQuality is called to determine whether the set threshold for good data % at each reference point was passed. This method replaces scores at reference points that do not pass the threshold with ‘NaN’ values.

QC 2 (assessment of all scores) :
Evaluation of whether there are a sufficient number of ‘good’ or ‘valid’ scores (non NaN values) after the scores are calculated and QC 1 is completed. If the percentage of resulting scores does not meet goodScoreThreshold, all scores are replaced with NaN values. The percent is calculated by:

  # of non-NaN scores / # possible scores * 100

For heidke and brier, and RPSS only the total category is evaluated in the assessment of % good scores. If the % of good scores satisfies the required score threshold, all scores are kept as is. Otherwise, all the scores for the total category as well as for the separate categories are replaced with 'NaN' values, except for RPSS - separate categories are not QC-ed, meaning there is no replacement of original scores for the separate categories regardless of the results of the total category QC.

It should be noted that both in both QC1 and QC2 each forecast source is evaluated separately.

Quality Control for reliability score

Because of the inherent properties of the reliability score, a unique method of performing QC is done that is a sort of hybrid between QC1 and QC2. In the Stats.java this occurs under the block of code labeled 'QC 2 Section' For reliability, the total number of forecast-observation pairs over all the categories (total categories) are assessed for each forecast source to determine the percent of good data used to create all the scores for each forecast source. Only the total category is assessed. If the total category QC does not pass, then the separate categories will also be considered a QC fail and all scores will be removed (changed to NaN values)

First the percent of good scores are calculated in Stats.java (instead of QCLibrary.countGoodScores like the other scores), across all points in both time and space. The equation to calculate this using "withEC" is (calculated in Stats.java for reliability):

  % good data = # of non-NaN fcst-obs pairs / # of expected fcst-obs pairs * 100

where the forecast-observation pairs are counted over all time and space data points. The equation to calculate the # expected fcst-obs pairs using "withEC" is (Stats.java for reliability):
For Total categories :
  # expected fcst-obs pairs = # dates * # locations * 3 (# categories)
For separate categories :
  # expected fcst-obs pairs = # dates * # locations

and "noEC" :
For Total categories :
  # expected fcst-obs pairs = ((# dates * # locations) - # EC forecasts) * 3 (# categories)

For separate categories :
  # expected fcst-obs pairs = ((# dates * # locations) - # EC forecasts)

Then it calls QCLibrary.removeAllScoresIfBelowQCThreshold to determine if the percentage of good fcst-obs pairs over all dates and locations combined does not meet the goodScoreThresholdReliability. If this QC is not passed, all scores are replaced with NaN.

Calculating the number of expected values (for all scores)

The # of expected dates are retrieved from the data object (# of unique dates associated with the retrieved forecasts) in all cases except for when the forecast type is ‘extendedRange’ and the forecast source is ‘manual’. In this situation, the number of weekdays are retrieved for the range of dates (for ERF manual, weekends are not expected to have forecasts).

The # of expected locations are retrieved from the forecast data (unique locations only counted, not duplicates), except for when the spatial type is ‘station’. In this case, a reference table in the database is used to look up an expected number of stations for that particular forecast setting. These values, however, are really based on the associated observation datasets as well since the dataset with the lesser stations would have to be utilized since only valid fcst-obs pairs are considered (not the forecast separately than the obs). This reference table was utilized because there are differences between forecast sources regarding how many stations to expect as well as how many stations are in the associated observations datasets. These values could change over time depending on the amount of stations in the forecast and observation datasets. These values can be viewed in the Google spreadsheet ‘dataSettings’ in the Verification > Data Google Folder :
https://docs.google.com/a/noaa.gov/spreadsheet/ccc?key=0Ao4v21vizXv0dE9fdkE0ZjZjUDJzNW1vWVhMbGY1M0E#gid=1

Score Quality Control by Category

Quality control for both types of QC (by each score and total score evaluation) for separate categories is done by the results of assessing the total category for all scores except RPSS, which does not evaluate or impact the separate categories at all. For each reference point (QC 1) if a score does not pass QC for the total category it is replaced with NaN and the scores for each of the separate categories are also thrown out (replaced with NaN), except for RPSS where no QC is done for separate categories (due to the nature of the score). For QC 2, if the percentage of good scores do not pass QC for the total category, all total category scores are replaced with NaN and all scores for each of the separate categories are also thrown out. As a result, plots may be blank and ASCII files may have many NaNs. This is intentional because it is assumed that the data is not “good” enough to produce results.

The counts/percents of good scores are printed in the header in the ascii files (some variety based on settings). The percents of good fcst-obs pairs data for each individual score are printed in a column in the ascii files with the associated score, where each row pertains to a score for a specific date or location. For reliability, all percents are set to NaN since they are also undefined. For reliability, the number of valid forecasts for each probability bin are printed in a column in the ascii file or in a table for the web application, since this information corresponds to what would be shown in a histogram with the reliability diagram.

Score Quality Control Methods and Arrays

In QCLibrary, the following methods perform important QC functions:

- removeScoresWithLowQuality - Performs QC 1 for non-reliability scores by replacing (removing) a score for a specific reference point.
- removeAllScoresIfBelowQCThreshold - Performs QC for non-reliability scores by replacing (removing) all scores in the score array.
- removeAllScoresIfBelowQCThresholdReliability - Performs QC for reliability by replacing (removing) all scores in the score array with ‘NaN’ values if the threshold for QC is not satisfied.
- countGoodDataPairs - For non reliability, this method counts the number of valid fcst-ob pairs (non NaN values) that makes up each score and the percentage of valid fcst-ob pairs making up each score. The counts are in variable arrays goodDataCountArray and the percents are in goodDataPercentArray.
- CountGoodScores - Returns the counts and percentages of scores that passed qc. The counts are in goodScoreCountArray and the percents are in goodScorePercentArray. The percentage is the number of non-NaN fcst-obs pairs (the count) to the total number of expected fcst-obs pairs. The total number of expected fcst-ob pairs is numExpectedFcstsPerScore for each individual score and numExpectedScores is an array of the expected number of scores for each forecast source.

Appendix H : Exception Package

When to Throw Exceptions

Here's when we should be throwing an exception:

- When we call logger.fatal(). This is because if we're calling logger.fatal(), we must want the process to stop execution.
- When something happens that would prevent the rest of the program from completing successfully.

Essentially, every time a fatal error is encountered, three things should happen:

1. logger.fatal is called with detailed error information (more appropriate for a developer)
2. Log.fatal is called with less detailed information (more appropriate for an end user)
3. An exception is thrown

Propagating an Exception Upward

In both the static and dynamic version of the Verification System, when an Exception is thrown, it gets propagated upward to the very top (applet for the dynamic version, driver for the static version). This happens because when an Exception is caught anywhere in the code (eg. method A), it then gets thrown, which cause the parent method (method that called method A) to catch the Exception, and in turn throw it. This process is repeated up to the very top level. At this point the Exception is caught, and the process is terminated.

For more detailed information about how Exceptions are used, see https://cpc-devtools.ncep.noaa.gov/trac/projects/Verif_System/wiki/CodingTeam/ThrowingExceptions

Appendix I: Logging

There are two ways to perform logging in the verification tool:

1. The web application display logging enables logs to get displayed to the VWT web application display in the “Warnings and Errors” panel. This utilizes a Log.java class at the core, which was developed by the verification system team.
2. 3rd party Apache Log4J logging software logs to :
    a. /logs/static.log file in the direct-access database command-line run version during run-time.
    b. Client’s Java console and Server Tomcat log area in the web application version during run-time. Different parts of the process logs to either the client’s or server’s log area depending on where the running occurs during that part of the processing.
    c. Client’s local /logs/static.logs and Server log area in Tomcat in the command-line web service database access version during run-time.
    d. Client’s terminal if initiated manually on a command-line. Similar to the other aforementioned way logging works when using web services, if web services is utilized and the process is initiated manually on a command-line then only client side processing logs will be seen printed to the terminal. If direct-access to the database method is used, all logs (with the appropriate log level(s)) are printed to the terminal in addition to the static.log file. The messages printed to the terminal would be the same as the content in static.log.

Depending on the type of message you want logged, you would call different log methods. You should always log to the log file, even if you are also logging to the web application version. Typically the message that gets displayed on the web application should be general enough for a user to understand, but detailed enough so that when the user reports the error, the maintainers of the verification system understands the bug well enough to debug. It should also be noted that for a client to view the logs in the Java Console, their Java preferences must have the console enabled and displayed.

This section will discuss how the two types of logging software works and how developers should utilize these methods.

How to view logs

As mentioned in the beginning of the logging section, the method of how the verification software is ran determines where the logging gets output. Below are details of how to view these various logs.

Web application display panel

After run-time is complete, the process would display any necessary web application method logs to the error panel on the applet. These logs utilize the Log.java class created by the verification system team that enables logs to get thrown upward to the web application.

Java Console

During run-time of the web-application, log4j will print logs that occur during processing on the client's machine to the user's Java Console. The console must be enabled and displayed by using the Java preferences. By default this console is not displayed to the user on their machine. This is typically used for debugging by retrieving detailed logging unnecessary to most users. This does not include logs from the parts of the process that occurs on the server associated with the SOAP services.

Server Tomcat log files

If the web services database access method is used, in either the web application or command-line initiation of the process, follow these instructions to view the log files:

Step 1 – SSH to the Tomcat server

```
ssh linuxName@wwwdev1.ncep.noaa.gov
```

where linuxName is your Linux username

Step 2 – Change into the Tomcat logs directory

```
cd /var/log/tomcat5
```

Step 3 – Open log file

```
vi catalina.out
```

Process-local log file

If the command-line direct-access database method is utilized, all the logs from log4j for the entire run-time process will get written to the /logs/static.log file in the verification system software directory.

If the command-line web services database access method is utilized, the parts of the process up to where the getResults() is and after the results object is returned, gets written to the client's machine in /logs/static.log. This is processing that occurs on the client's machine, and does not include logs during the processing that occurs on the server, behind the SOAP services. The logs from the part of the processing that occurs on the server gets written to the Tomcat log area (see above section).

Apache log4j logging

This is utilized for both the web application and command-line logging. For the web application, these logs get sent to either the Tomcat server or the client's Java console depending on where the part of the process runs. For the command-line run direct-access database method, logs get written to a /logs/static.log file. The command-line web service database access method writes to the client's /logs/static.log file if the processing occurs on the client's machine and to the Tomcat log area if processing occurs on the server.

The Apache log4j software file was included in the verification software by the developers, in /library/java/log4j.jar.

About log4j configuration files

The applet and the command-line methods set the configuration files that get used for the logging differently. The main option you would want to set is the level of logging to print during run-time. These levels are described in the 'log4j log levels' section. The PropertyConfigurator class must be first utilized in the initialization classes before usage in the rest of the downstream software code. The configuration files are located in the /input directory that contains the settings for the log4j logging software, including the level of logging desired during run-time. There are 2 log configuration files:

- logConfigApplet.txt - contains log4j log settings used by the web application version during run-time.
- logConfig.txt - contains log settings used by the command-line run version during run-time and prints to log files, to both the client and/or server side.

In these configuration files there are settings for the console (stdout) logging, files (on client's machine and Tomcat server log area), or Java console. If you notice that the logging levels are not working properly, change all instances of log level settings to the same log level, since developers were somewhat uncertain about this. These lines contains the options :

```
log4j.appender.stdout.Threshold=DEBUG
.
.
log4j.appender.file.Threshold=DEBUG
```

Other specifications such as what log file to output to, and layout patterns are also set in this file. The online Apache log4j resources should be utilized for further documentation that this document does not cover.

Setting log4j configuration file in applet
The configuration file gets set and is hard-wired currently in the ChartApplet.java and SpatialApplet.java classes. The necessary steps are done In the applet init() method :

First, the log4j property configuration package is imported at the top of the class:

```
import org.apache.log4j.PropertyConfigurator;
```

Then the logger configuration is set up:

```
//----------------------------------------------
// Set up the logger
//----------------------------------------------
// Set up configuration file
String configFile = "/input/logConfigApplet.txt";
// Get the URL version of the file
URL url = ChartApplet.class.getResource(configFile);
// Configure the logger
PropertyConfigurator.configure(url);
```

Setting log4j configuration file in StaticRunDriver.java
There is a setting in the build file (build.xml) that contains the 'run_command_line_direct_acces' and 'run_command_line_web_services' targets that contains the name of the log configuration file to use. This gets accessed when these targets are used to initiate the command-line version of running. Currently, log4j is configured to use the /input/logConfig.txt file for command-line logging. The lines in the build.xml are:

```
<!-- Runs the Verification System in "Command Line Web Services" -->
  <target name="run_command_line_web_services" depends="" description="Runs the Verification System in 'Command Line Web Services' Mode (see README)">
     <!-- Check for $VERIF_HOME environment variable -->
     <check_for_env_var var="VERIF_HOME" />
     <java classname="gov.noaa.ncep.cpc.services.WebServiceRunDriver"
         fork="true"
         failonerror="true">
           <arg value="-finput/settings.xml"/>
           <arg value="-linput/logConfig.txt"/>
     <classpath>
           <fileset dir="${library}/java" includes="**/*.jar"/>
           <fileset dir="${build}" includes="verif_client.jar"/>
     </classpath>
     </java>
</target>
```

Using log4j for logging

Now that the log configuration is set, below are steps on how to use the log4j logging software.

1. Import the log4j logger software at the top of the class :
   `import org.apache.log4j.Logger;`

2. Declare the logger variable first in the constructor :
   static Logger logger; (Declare this variable at the top in the constructor)

3. Initialize the logger. This must be done in every class before usage. The initialization classes initialize the logger a little differently than the other regular classes.

   This is done early in the init() method in the applet and in the main() method of StaticRunDriver.java :
   logger = Logger.getLogger(StaticRunDriver.class.getName());

   In the other regular non-initialization classes, initialize the logger in the constructor methods. It is typically done in the constructor so the code can fully utilize logging as soon as possible :

   logger = Logger.getLogger(gov.noaa.ncep.cpc.data.Data.class);

4. Now, log4j logging can be utilized anywhere in that class. To add a log message, use the logger object, followed by the assigned level of the message and the message formatted as a string in parenthesis. For example :
   logger.fatal("Could not load forecast data for forecast sources. " + e);

   'e' is utilized in this example, which would be an exception in a try/catch block. The level that you assign it will determine whether it will get written to the log file/error console, which depends on the level of logging that is set in the log configuration file. Log messages with levels of the set log level in the config file and above that level will get logged (logger inheritance). Below is the logger hierarchy in the order from lowest to highest level of logging :

   TRACE < DEBUG < INFO < WARN < ERROR < FATAL

   For example, if the logConfig.txt is set to 'WARN', then all the messages with the levels WARN, ERROR, and FATAL will get logged. So, when assigning a level to a log message, you must think about the intention of the message and at what level you would want the log to be displayed at. The various levels are described in the next section, 'log4j levels'.

log4j log levels

The below outlines the various levels of logging that can be utilized. It is presented in order from including the lowest to highest level of logging.

TRACE - The TRACE Level designates finer-grained informational events than the DEBUG
DEBUG - The DEBUG Level designates fine-grained informational events that are most useful to debug an application.
INFO - The INFO level designates informational messages that highlight the progress of the application at coarse-grained level.
WARM - The WARN level designates potentially harmful situations.
ERROR - The ERROR level designates error events that might still allow the application to continue running.
FATAL - The FATAL level designates very severe error events that will presumably lead the

application to abort.

Web Application Display Logging

A solution was designed by the verification software team to enable logs anywhere in Java code to be displayed to the web application error panel if it is used during run-time. Below is information regarding the implementation and ‘how to use’ information.

Front-end

In the verification software, sub-directory /web/library contains the Javascript library Messaging.js. This contains the functions used in this implementation. There are two methods in Messaging.js, “appendToPanel” and “popup”.

The “appendToPanel” function displays a passed HTML formatted string to a div tag on the web page, which is also passed to this function. This function accepts an object that contains a string containing 2 pieces of information, separated by a semi-colon. The first part of the string (before the semi-colon) contains the HTML formatted String to display on the panel, and the second part of the string (after the semi-colon) contains the name of the div tag to append/display to. The String passed would be formatted as "$error ; $divName" where $error is the HTML formatted string to append and $divName is the DIV id to which the HTML should be appended. In the verification software, currently the div tag to append to is set to ‘errorPanelText’ . This div ID is in /web/index.php, the PHP page that contains the display of the tool. Logging to this panel should be used for non-fatal messages, such as errors or warnings. A parsing method is used to separate this information and display a message accordingly.
When the ‘popup’ function is called, a pop-up is displayed to the user on the browser with a passed HTML formatted string message. This function accepts an object containing one piece of information, an HTML formatted string of a message to display.
Logging to the pop-up should be used for fatal messages that would typically stop or indicates that the chart/results cannot be retrieved. The user will see a pop-up box with the passed error and must close the box to continue to re-submit the form in the verification web tool.
Messaging.js also has a method, ‘clearPanel’ that can be called to empty the contents of a passed div tag. This is called by the Log.java method empty(), which is described in the Java section below.
Currently, the index.php file that contains the applet has the following syntax that sets up the panel:

```
<!-- Error Panel -->
<div id="errorPanelContainer">
        <p class="title">Warnings and Errors</p>
        <hr>
        <div id="errorPanelText">
                <!-- <p class="title">Warnings and Errors</p>
                <hr> -->
                <p class="content"></p>
        </div>
</div>
```

The div id “errorPanelContainer” contains the title, as well as a nested “errorPanelText” div id. This is the panel that gets cleared and logged to. The reason why these are nested is so that when the content area of “errorPanel” is emptied, the title remains.
The look and feel is controlled by the associated div tag CSS styles set in the /web/styles/index.css file that contains display information for each of these div ids. This makes the entire area look like one panel separated by a horizontal rule for the title and content.

Java (back-end)

The class handling the web application display functionality is in Log.java, located in the ‘qc’ package of the verification system software. This utilizes the JSObject (Javascript object) to call the Javascript Messaging.js functions. JSObject software is Netscape technology, which can be easily downloaded for free online and incorporated into the available classes of the web software used. The verification system software already includes this in the /library/java/JSObjects.jar file.

The methods in the Log.java class handles the passing of log messages from any Java code in the software to the web error panel display. Once the JSObject window has been set up in the code, any subsequent code following the setup can easily call this functionality (including the applet) right after the window is set. The logging syntax was set up to be similar to the Apache Log4j format for usage ease. Short class and method names were used to mimic log4j logging usage and for quick logging throughout the code.

Using the web application display for logging

Below are the steps to follow to utilize web application display logging:

1. In the applet, the Javascript object technology must be imported at the top:
   import netscape.javascript.JSObject; //At top of code
2. Import the qc package at the top of the class, which contains the Log.java class: import gov.noaa.ncep.cpc.qc.*;
3. In the init() method of the applets, the window must be retrieved for calling Javascript later. This line is set in the applet relatively early so that logging is set up as early as possible for usage in the rest of the process. The line in the applet code is :
   window = JSObject.getWindow(this);
4. Then the JSObject can be set in the Log.java class. This is done in this class so that the window object is accessible to the logging methods whenever they are called from other classes and methods. Right after the above line where the JSObject window is retrieved, set the JSObject in the Log.java class for it to reference throughout run-time:
   Log.setJSObject(JSObject window);
5. To make sure that the messages clear for each run of the applet, the window is cleared by calling the empty() Log method in init() method after setting the JSObject, and the beginning of the update() method in the applet:
   Log.empty(String div);
   Where div is the div tag area to clear on the web page of messaging. In the verification

web tool the div tag to clear is “errorPanelText”. This is in the PHP file that contains the applet. Therefore, in the Java code, a web panel log would look like: Log.empty(“Invalid output...”,”#errorPanelText”);

6. Now the Log() method can be called to add an error to any class, including in the applet after the JSObject has been set. See the below section on ‘How to use logging after implemented in code’ for how to log once set-up in the code.

About the Log.java methods

The error()/warning() and fatal() methods in Log.java both perform similar functions. They both :

1. Accept a String(s) and convert it to a object, which can be passed to the Javascript Messaging.js class.
2. Perform default formatting is performed on the passed HTML string, currently a “>” character is concatenated to the beginning of the passed message for bulleting format purposes. A <br> HTML tag is also concatenated to the end of the passed message so that each passed message is displayed on a separate line in the display panel on the web page. Because there is a <br> tag, it is suggested that a <p> tag is not wrapped around a passed error message, this would result in large spacing between lines.
3. Convert the message string to an object that can be passed to the Javascript method.
4. Use the window object (set by the setJSObject() method in this class in the applet), to call an appropriate method in the Javascript Messaging.js class to display the message.

Difference in logging methods:

In the error()/warning() methods, the passed error message string and div string (to display to) are concatenated and separated by a semi-colon before being converted to an object to pass to the Messaging.js class. This is because JSObject interaction between Java and Javascript can only pass one object from Java to Javascript (as of we know). This method will call the appendToPanel() function in Messaging.js.
The fatal() method simply converts the passed String log message to an object before passing it to Messaging.js. This method will call the popup() function in Messaging.js.
Now the Log.java class can properly access the JSObject window to call functions in Messaging.js to display a log message.
The method that is called by the applet to clear the messaging panel at each beginning of a run of the applet is empty(String divTag) in Log.java. This method needs to be called at the beginning of the init() method after setting the JSObject and the beginning of the update() method in the applet so that the panel is cleared at the beginning of run time and also before any updates are made to re-run the process.

This empty() method works by passing a String of the name of the div tag to clear in the PHP page that contains the messaging panel (in verification web tool this div tag is “errorPanelText”). When this method is called, the div tag string is converted to an object that is passed to the Javascript messaging class “clearPanel”. Then clearPanel uses a JQuery command to empty the passed div tag.

Logging non-fatal errors (error/warning) to a web panel

When creating a non-fatal log message on the web application display, use error or warning for non-fatal errors. These will not impede the final results and these errors do not necessarily lead to a failure in completed processing. These messages get displayed on the error panel of the web application in the client's browser. In many of the messages, the wording typically includes what the client/user should report in case of an error or warning. This would help users provide useful information to the maintainers of this software. To create a log message, call the error() method in Log.java, and pass two Strings :

```
Log.error(String htmlString, String cssSelector)
```

where htmlString is an HTML formatted error string to display, and cssSelector is a String representing the CSS selector of the page element to display the error on (see http://www.w3schools.com/cssref/css_selectors.asp for help with CSS selectors). Currently the index.php in the verification tool contains the 'errorPanelText' <section> that is setup to display messages (CSS selector is "#errorPanelText"). Below are examples for the error and warning messages :

```
Log.error("Could not format...Report formatting error...","#errorPanelText");
Log.warning("Could not load data...Report data error...","#errorPanelText");
```

Below describes what would get displayed in either an error or warning message :

	Text color	Preceding symbol	Usage
Warning	yellow	[!]	Something may be wrong but results will still be displayed
Error	red	[x]	Something went wrong but some or all of the results will be displayed

Logging fatal errors (fatal) to a web panel

The fatal level should be used for fatal errors. These are errors that prevent successful completion of the processing. This will result in a pop-up window in the browser, rather than a display on the error panel. The user is required to close this popup to continue and re-submit the form to process another selection of settings. Below is an example of how to call the fatal level of logging:

```
Log.fatal("Data corrupted...Report data error...","#errorPanelText");
```

How to format log messages for web application display :

- The String message that you pass should not be wrapped in a <p> tag because by default, all messages will have a <br> tag concatenated to the end of the message so that all new log lines are on separate lines.

- If you want to include a Java command or information included in the message, create a concatenated String with that info and then pass the string as the first String argument to Log.error. For example, you want to include the value of output type in the error string displayed:

```
String error = settingsObj.getOutputType() + " is not a valid outputType!";
Log.error(error,"#errorPanelText");// "#errorPanelText" is the div tag to display to.
```

Appendix J : Verification of uneven terciles / extremes

This software was originally designed for assuming only even, symmetric terciles would be verified, with tercile thresholds at the 33% and 67%. Because this code handles the logic assuming three categories, to extend the verification functionality to non-even tercile forecasts, the database table structure and code has been updated to still expect three categories of a forecast, but the thresholds of those terciles can be flexible, as long as it is an acceptable unit accepted by the SettingsHashLibrary.java. This documentation and the code refers to forecasts that do not have even, equal terciles as “uneven terciles”. This enables extremes to be verified. Even if you are only interested in processing one category (e.g. tmax < 15th percentile), you must import data so that there are 3 categories. For example:

- lower category (“B”) : tmax < 15th percentile
- middle category (“N”) : 15th percentile =< tmax <= 85th percentile
- upper category (“A”) : tmax > 85th percentile

To verify extremes, the lower and upper categories would typically be symmetric in terms of percentile width, with the middle category containing the remaining portion of percentiles between those thresholds. The software settings validation currently allows category units to be full fields in addition to percentiles (e.g. degF), but the software has not been tested or adapted to process this type of data yet. Regardless, Heidke and Brier skill scores cannot be calculated for these because of the inherent nature of these quantities that require knowing how climatology would be able to forecast correctly.

Settings for uneven terciles/extremes

The only difference between normal even terciles and uneven terciles in the settings XML file is the variable setting. The format for the ‘variable’ tag is:

${variable}-${categoryUnit}-${thresholdLower}-and-${thresholdUpper}

For example, for maximum temperature evaluating percentiles at the 15th and 85th percentiles the snippet of settings.xml would look like:

<variable>tmax-ptile-15-and-85</variable>

for tmax < 32degF and > 100degF:

<variable>tmax-degF-32-and-100</variable>

Databases for uneven terciles/extremes

The table name is formatted so that it contains the category unit and thresholds information (in addition to the forecast source and other information). An example of the table name for minimum daily temperature at the day 8 lead, 2-degree gridded, for the GEFS reforecast calibrated tool at the 15th and 85th percentile is :

tmin-ptile-15-and-85_rfcstCalProb_gefs_00z_08d_01d_grid2deg

Logic supporting uneven terciles/extremes

Data.java - Category selection

- Data.java determines from the input forecast probabilities of each category, which is the favored category for verification.
- Methodology assumes that 20% is the minimum probability threshold to deem a forecast probability as the favored category. This probability was selected based on the fact that CPC decided 20% was the minimum probability to consider an extreme to have a slight chance, which would potentially make it to the official probabilistic hazards forecast map.
- For uneven terciles, the favored forecast category is selected based on the following logic:
    - If both the lower and upper categories are less than 20% probability, pick the middle category.
    - Else if it is long range, and all 3 categories have equal probabilities pick “EC” category.
    - Else pick the greater probability of the lower or upper category.
- For even terciles, the minimum threshold probability is 33%, rather than 20%.

PlotChart.java - Display labels

- updateLabels() determines the minimum possible Heidke HSS value for the charts.
- Minimum value for HSS calculated by :
    - *-p/(100-p)-10*, where p is the probability window associated with a percentile

(probability that climatology is expected to get forecasts correct). p is obtained from the variable setting by calling SettingsHashLibrary.getPercentileWindows(), which determines it by:

- lower category = thresholdLower
- middle category = 100.0f - ((100.0f - thresholdUpper) + thresholdLower)
- upper category = 100.0f - thresholdUpper

where thresholdLower and thresholdUpper are the thresholds included in the percentile specifications in the variable name. Note: This does not apply

- If separate or total categories are processed, the greatest probability window is the ‘p’ used to calculate the minimum HSS value (*-p/(100-p)-10)*.
- If an individual category is processed, use the appropriate percentile window of the selected category as the p value to calculate the min HSS.

FormatLibrary.java and WriteLibrary.java

- The category label is retrieved from SettingsHashLibrary.getCategoryLabel() so that the XML that creates the chart contains the appropriate information for the legend. The labels are based on the category unit.
- Formatting is adapted for uneven terciles for chart and ASCII output, including header labels.

StatsLibrary.java

Heidke Skill Score and Brier Skill Score needs adjustment for uneven vs. even terciles.

- The expected probability of climatology getting forecasts correct is based on the percentile. The window of probability is the same as the percentile window. For the example of the 15th and 85th percentile thresholds, the expected probability windows for the lower, middle, and upper categories are 15%, 70%, and 15%. This adjustment is applied to both HSS and BSS.
- The total Heidke score is calculated as a weighted mean. The total HSS is calculated similarly for non-even and even terciles as :
  - $H_T = (H_B * N_B + H_N * N_N + H_A * N_A) / (N_B + N_N + N_A)$

  where H is HSS of each category, N is the number of forecasts with the subscript denoted category, and subscripts B, N, and A, represent thte 3 categories, and T is total categories. For uneven terciles, **only the lower and upper categories are included.** For extremes, you would not be interested in including the middle category.

Appendix K : Configuration Options

There are a set of configuration (with extension .conf) files for both the database settings and other more general settings. Depending on how you run this process (using the web services vs. directly accessing the database) different .conf files are accessed by the process. The safest method is to set the options the same in both config files. These configuration files listed only refer to the ones that directly impact the processing. The log configuration files are described in Appendix I: Logging

Configuration files used for different ways of processing :

1. Web services (command line and web application) - Uses verif_server.conf and

verif_client.conf.
2. Command-line run accessing database directly - Uses verif_direct_access.conf only. It should be noted that this version requires that the software be located on the same machine as the database.
3. verif_data.conf - Used for processing of data importing that occurs upstream of the verification processing.

Description of Configuration files :

- verif_client.conf - Contains the URL of the web services and quality control threshold settings.
- verif_server.conf - Configuration file with generic settings that resides on the server's side. If the web version or Servlet version of command-line is ran, both this and the verif_client.conf is accessed.

  Contains the database and quality control threshold settings.
- verif_data.conf - Data import configuration file. These settings apply to the part of the process that ingests forecast and observation data from flat files into the database as well as other database maintenance functions. This is ran prior to verification processing or prior to when users would use the web tool. This is not accessed during the processing of the verification software, only during the data import/management functions.

  Contains database and email contact settings.

For example, verif_data.conf - looks like (pared down, without comments in actual file):

```
#------------------------------------------------------------------#
#     Verification System Data Import Configuration File      #
#------------------------------------------------------------------#
[mysql]
host         = vp-cpccfmysql.ncep.noaa.gov
user         = first.last
password     = MyPassword
db_fcst      = cpc_forecasts
db_obs       = cpc_observations
db_ref       = cpc_reference
db_vwt       =
dateFormat   = yyyy-MM-dd

[email]
toEmail   = first.last@noaa.gov
subject   =
fromEmail = cpc_processName@noaa.gov
fromName  = Process Name
logLevel  = ERROR
```

The top section contains MySQL settings, such as the host, user, and password. Typically the other settings (db_fcst, db_obs, etc.) would stay the same as what is already set. The bottom section contains other settings, such as the email contacts and logging level for the Perl scripts in the import data process.

Many of the other listed .conf files have a similar set up. Typically it is the database host, user, password, URL of web services, and QC thresholds you would want to change.

Setting the Quality Control Thresholds

The .conf files that contain QC thresholds have a section with a header '[qc]'. There are two settings, goodDataThreshold, goodScoreThreshold, and goodScoreThresholdReliability.

See the quality control section in Appendix G for more details.

Appendix L : Results Object

The results object was necessary to follow the Java Bean convention to work in the framework with SOAP web services to enable score and score information data to be passed between the client and server. The results object encapsulates this data and allows information to be passed between the client and server in a secure way. More information regarding the Results object in the context of the software is in the section 'Process Flow for the Web Application version'. Below are descriptions provided by Damian Hammond at University of Arizona (since he developed most of the results object).

About the Results Object

One of the main goals was to separate the applets from making direct calls to the database. Regardless of the mechanism of this separation the applets still needed the data found in the Stats and Data objects. The Stats object was converted to fit the conventions needed to make it a java bean. This allows it to be deconstructed and reconstructed by frameworks which would provide the communication between the client and server. There was an attempt to also convert the Data object to a bean but it was eventually decided that the raw data in the Data object was not needed and could be quite large. The Data object however did have data elements that were needed. Rather than significantly modify the Stats and Data objects the Results bean was created to encapsulate the Stats object and the specific data elements needed from the Data object.

VerificationDriver.getResults(...) , VerificationSystemServices.getResults(...) , and ServiceCallThread.callGetResultsWebService(...) all return a Results object but that does mean they belong to the Results object. We could make a design decision to move those methods to the Results class, in which case they would then belong to the Results object. That however would cause some complications. There are certain reasons why I did not do so and I will try to explain those here. It is always an option to do so if you feel that would make things easier to follow and the time is available to do so.

About VerificationDriver.getResults(...)

I made as few changes to this class as I could because it seemed one of the main drivers for the data access and was key in the command line tool you had put together with StaticRunDriver. If I remember correctly the main change was the introduction of the Results object and the ability to request this object after VerificationDriver.runDriver(...) had been called. One of the key factors here is that the VerificationDriver required the database connection to be created by the calling entity. StaticRunDriver for instance accesses a configuration file that provides info on the database to use.

About VerificationSystemServices.getResults(...)

The VerificationSystemServices class is designated by input/services.xml as the class that will define the SOAP web services for the verification tool. Any method added to this class will become an available service call. Note the helloWorld() method that is still in place. If you look at the deployed services you will see that helloWorld is an available service. VerificationSystemServices.getResults(...), thru the axis2 framework, becomes our main service that the applets use to get the data they need. Another important thing to note here is that the only reason that getResults(...) can return the Results object is because it follows the java bean conventions. This allows the axis2 framework the ability to know how to work with it. Like the StaticRunDriver this method has specific data as to what database to use, and after creating the database connection, it will use the VerificationDriver in a way similar to the StaticRunDriver.

About ServiceCallThread.callGetResultsWebService(...)

This is similar to VerificationDriver.getResults(...) except that the underlying mechanics are different. Where use of the VerificationDriver accesses the database directly, this method calls the web service provided by the VerifictationSystemServices and the axis2 framework.

About multiple instances of getResults()

There are two getResults() methods, belonging to the Results object: one in ServiceCallThread.java which gets called in the situation that web services is used, and one in VerificationDriver() in the case that the direct-access database version is used (typically command-line run verification), and can directly pass back that results object from the VerificationDriver, ie. to the StaticRunDriver. There are various methods throughout the software that utilize or return a Results object.

Both VerificationDriver.getResults(...) and ServiceCallThread.callGetResultsWebService(...) could be moved to the Results class. I would even be tempted to combine both those methods into one in but that would take some time.

Appendix M: 3rd Party Libraries Used

1. */library/java*
    - commons-io-1.4.jar – Apache set of classes
    - mysql.jar – MySQL classes
    - xml-apis.jar – XML API classes
    - xercesImpl.jar – Xerces XML parser formatting classes
    - log4j.jar – Apache logging classes
    - jcchart.jar[1] - JClass software, which contain classes that are utilized in this software.

[1]See JClass documentation in /docs/jcchart.pdf

- commons-cli-1.2.jar
- commons-lang-2.4.jar
- JSObjects.jar
- jsdoctoolkit-ant-task-1.0.2.jar
- js.1.7R2.jar
- jaropt.jar

2. */library/perl*
    - Stations592List.pm
    - SiteList.pm
    - Month.pm
    - ForecastDivisionList.pm
3. */library/jsdoc-toolkit*
    - Directories and files for Javadoc (documentation building)
4. Google Maps server - The public Google Maps Server is utilized to display results on the map on the web application.