Documentation for Ryan's model-evaluation code

How to read: Part 0 of this document describes the input files for model evaluation, each containing the predictions and targets for one valid time.

Part I of this document describes evaluation code that does *not* stratify temporally (*e.g.,* by hour-of-day or month-of-year). Thus, all evaluation code described in Part I stratifies either spatially (by NBM grid point) or not at all. "No stratification" means that evaluation metrics are computed over the entire dataset, with no regard for geographic location or time.

Part II of this document describes code that *does* stratify temporally, by either hour-of-day or month-of-year. Additionally, some code described in Part II can stratify *spatio*temporally – either (1) by NBM grid point and hour-of-day or (2) by NBM grid point and month-of-year.

The GitHub repository is here: https://github.com/thunderhoser/ml_for_national_blend

Part 0: Input files

Each input file is a NetCDF, containing the predictions and targets for one valid time. Dimensions in the NetCDF file are as follows:

- `grid_row`: Indexes over rows in the spatial grid.
- `grid_column`: Indexes over columns in the spatial grid.
- `field`: Indexes over target fields.
- `ensemble_member`: Indexes over ensemble members produced by the neural-net model.

Variables in the NetCDF file are as follows:

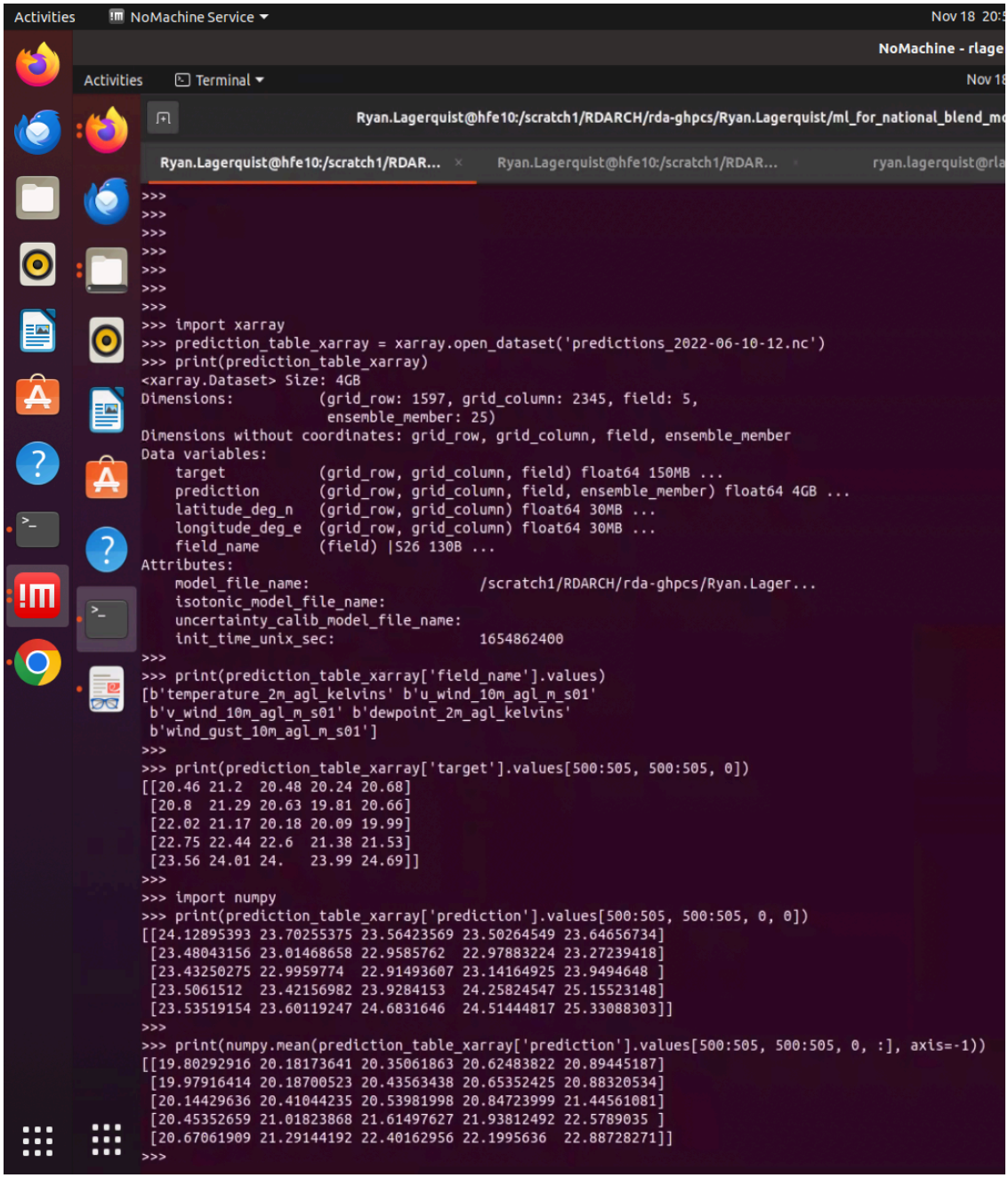
- `field_name`: numpy array with dimensions `(field,)`, where the k^{th} element is the name of the k^{th} target field. Field names also contain units – *e.g.,* `wind_gust_10m_agl_m_s01` is "wind gust at 10 metres above ground level in metres per second". However, note that the units for temperature and dewpoint are a lie. The field names are `temperature_2m_agl_kelvins` and `dewpoint_2m_agl_kelvins`, but values in the file (stored in the variables target and prediction) are in degrees Celsius.
- `target`: numpy array with dimensions `(grid_row, grid_column, field)`, containing target values. My target values ("ground truth") come from the URMA dataset.
- `prediction`: numpy array with dimensions `(grid_row, grid_column, field, ensemble_member)`, containing predicted values.

- `latitude_deg_n`: numpy array with dimensions `(grid_row, grid_column)`, containing the latitude at every grid point in degrees north.
- `longitude_deg_e`: numpy array with dimensions `(grid_row, grid_column)`, containing the longitude at every grid point in degrees east.

My evaluation code makes no assumption about the grid. In other words, my code can handle any grid specified by `latitude_deg_n` and `longitude_deg_e`. The one exception *might* be grids that straddle the International Date Line – but as long as we're using a regional domain contained entirely within North America, that shouldn't become an issue.

The screenshot below shows you can open and play with a prediction file, using xarray. Since the files are NetCDF, you don't need to use xarray; you can use any NetCDF tool you want. The five print statements print the following:

1. A summary of the whole table
2. A list of target fields, in the order they appear in the target and prediction arrays
3. URMA ("actual") temperatures, from grid rows 500-504 and grid columns 500-504
4. Predicted temperatures from the 0th ensemble member, from grid rows 500-504 and grid columns 500-504
5. Predicted temperatures from the ensemble mean, from grid rows 500-504 and grid columns 500-504



Besides prediction files, the model-evaluation pipeline requires one more input file. Whenever a script has the input argument `input_target_norm_file_name`, it needs the file containing normalization parameters for the target variables. For each target variable *y,* this file contains several statistics based on the training data, including the mean value of *y*. For evaluation metrics that involve comparing to a climatological model – *i.e.,* a naïve model that always predicts the climatological mean – the mean value stored in this file is considered to be the "climatological" mean. You can find that file in the same folder as this documentation. You don't need to know anything about the file; you just need to feed it to some scripts as an input

argument. But if you're curious, the file is in NetCDF format, and the variable names within the file should make it self-explanatory.

Part I: No temporal stratification

Temporally agnostic model evaluation is handled by nine different Python scripts. These are described in turn.

1. evaluate_model.py

This script evaluates deterministic forecasts.[1] `evaluate_model.py` can run in both gridded and ungridded mode, controlled by the input argument `per_grid_cell`. In ungridded mode, `evaluate_model.py` produces one set of evaluation metrics for the whole spatial domain (NBM grid) and time period (specified by input arguments). In gridded mode, `evaluate_model.py` produces one set of evaluation metrics, averaged over the whole time period, at every grid point. So in other words, ungridded mode means that you're averaging over space, while gridded mode means that you're not. Currently, `evaluate_model.py` can handle only one lead time. However, it can handle multiple target fields (any combination of 2-m temperature, 2-m dewpoint, 10-m zonal wind, 10-m meridional wind, and 10-m wind gust) at the same time. `evaluate_model.py` computes the following statistics for every target field (or for every target field and grid point):

- Mean and standard deviation of actual values. You can think of this as the true climatology.
- Mean and standard deviation of predicted values. You can think of this as the predicted climatology.
- Mean squared error (MSE)
- MSE decomposed into two components: bias and variance
- MSE skill score (relative to climatology)
- Dual-weighted mean squared error (DWMSE)[2]
- DWMSE skill score (relative to climatology)
- Mean absolute error (MAE)
- MAE skill score (relative to climatology)
- Bias
- Bias in the spatial minimum. This is the average value, for every target field, of `min_prediction_over_entire_grid - min_actual_over_entire_grid`. Note that this statistic can be computed only per target field, not per target field per grid point

[1] When I say "deterministic forecast," I mean only the ensemble mean; when I say "probabilistic forecast," I mean the full ensemble; and when I say "uncertainty estimate," I mean some measure of ensemble spread, such as the standard deviation or X% confidence interval.
[2] See definition below.

(since it involves taking a min operation over *all* grid points). In other words, this statistic is computed only in ungridded mode, not in gridded mode.

- Bias in the spatial maximum. This is the average value, for every target field, of `max_prediction_over_entire_grid - max_actual_over_entire_grid`. Again, this statistic is computed only in ungridded mode, not in gridded mode.
- Correlation (specifically, the Pearson correlation between actual and predicted values)
- Kling-Gupta efficiency
- Reliability (as reported in the reliability diagram or attributes diagram; see Hsu and Murphy 1986 for more details)

DWMSE is similar to mean squared error, but with a twist that weights extreme values. The weight for every prediction/target pair is the maximum absolute value of the two. For temperature and dewpoint, the values in prediction files are always in °C; because DWMSE weights large *absolute* values, it weights values far below and above 0 °C more heavily. In other words, DWMSE, the way I have set it up, weights extreme cold/warm temperatures more heavily than "mundane" temperatures near 0 °C – and also weights extreme moist/dry dewpoints more heavily than "mundane" dewpoints near 0 °C.

The other target variables are 10-m zonal wind, 10-m meridional wind, and 10-m wind gust. For zonal and meridional wind, large wind speeds (whether positive or negative) are weighted more heavily than small wind speeds (near zero). For wind gust, large wind speeds (which can be only positive) are weighted more heavily than small wind speeds (near zero).

$$\begin{cases} \text{MSE} & = \frac{1}{N}\sum_{i=1}^{N}\left[y_i^{(\text{actual})} - y_i^{(\text{predicted})}\right]^2 \\ \text{DWMSE} & = \frac{1}{N}\sum_{i=1}^{N}\max\left\{|y_i^{(\text{actual})}|, |y_i^{(\text{predicted})}|\right\}\left[y_i^{(\text{actual})} - y_i^{(\text{predicted})}\right]^2 \end{cases}$$

Furthermore, `evaluate_model.py` computes all quantities needed to plot two figures for every target field:

- The attributes diagram (which is a reliability diagram with extra lines/polygons in the background; see Hsu and Murphy 1986)
- The Taylor diagram. This diagram shows the Pearson correlation, standard deviation of each dataset (actual and predicted), and centered RMSE, in the same polar plot. See Taylor (2001) for details.

`evaluate_model.py` has the following input arguments:

- `input_prediction_dir_name`: Path to directory with prediction files. Every prediction file contains both predicted and actual (URMA) values, which are the main inputs needed for model evaluation. Within this directory, the expected file-naming structure is "predictions_<yyyy-mm-dd-HH>.nc", where the <yyyy-mm-dd-HH> part is the forecast-initialization time.
- `init_time_limit_strings`: This argument specifies the evaluation period, *i.e.,* the time period over which all evaluation metrics will be computed. For example, if your evaluation period is the years 2021 and 2022, init_time_limit_strings would be a list with ("2021-01-01-00", "2022-12-31-23"). Either way, the list must be two items long, with time stamps in the format I just gave. **Please note that these time limits – and all other time limits required by my scripts as input arguments – are inclusive rather than exclusive.** Hence, the above list would include 0000 UTC 1 Jan 2021 and 2300 UTC 31 Dec 2022.
- `num_bootstrap_reps`: Number of replicates for bootstrap resampling. For example, if you want to bootstrap every evaluation metric 1000 times, make this 1000. Then all 1000 estimates for every metric will be written to the output file. Bootstrapping is good for estimating the uncertainty in an evaluation metric, which allows you to create confidence intervals (*e.g.,* error bars in a graph). However, bootstrapping is computationally expensive, so by default I set `num_bootstrap_reps` = 1 (which is equivalent to saying "no bootstrapping"). If it turns out that you really want confidence intervals (or to conduct a formal statistical-significance test), you can always go back and run the script with `num_bootstrap_reps` > 1.
- `target_field_names`: List of target fields for which you want to evaluate predictions. The valid names here are "temperature_2m_agl_kelvins", "dewpoint_2m_agl_kelvins", "u_wind_10m_agl_m_s01", "v_wind_10m_agl_m_s01", and "wind_gust_10m_agl_m_s01".
- `input_target_norm_file_name`: Path to normalization file for target variables. This file contains normalization parameters for every target variable. This is needed because some metrics computed by `evaluate_model.py` are skill scores, comparing a statistic (something like MSE) from the model with the same statistic obtained by "climatology" (a dumb model that always forecasts the climatological mean). To form the climatological model, we need this file to find the climo means.
- `num_relia_bins_by_target`: Number of bins in reliability diagram per target variable. This must be a list of length *T,* where *T* is the length of `target_field_names`.

- `min_relia_bin_edge_by_target` and `max_relia_bin_edge_by_target`: These arguments control the bin edges in the reliability diagram for every target variable.[3] Each argument must be a list of length *T,* where *T* is the length of `target_field_names`.
- `per_grid_cell`: Boolean flag. Make this 1 for gridded mode, 0 for ungridded mode.
- `keep_it_simple`: Boolean flag. Make this 0 to compute everything, including Kolgomorov-Smirnov stuff and all values needed for the reliability diagram.[4] Both K-S stuff and the reliability diagram take a long time to compute, which is why you might want to set `keep_it_simple` to 1. If you're running `evaluate_model.py` in ungridded mode, the computing time isn't really a big deal. But in gridded mode, you probably don't want to compute all the reliability stuff for every grid point (there are ~3.7M grid points in the NBM, so you're not going to look at the reliability diagram for every grid point anyways).
- `compute_ssrat` and `compute_ssrel`: These are Boolean flags. For now, just make them 0 and don't worry about them. Both flags are explained later in this documentation.
- `output_file_name_or_pattern`: Path to output file. Evaluation metrics will be saved here in NetCDF format.

Here is an example of how you would call the script from the command line (or from a Bash script that you can submit as a Slurm job). Note that even though the official names for temperature and dewpoint end in "_kelvins", the prediction files contain these variables in °C, so the units in `min_relia_bin_edge_by_target` and `max_relia_bin_edge_by_target` should be °C also. Sorry! The arguments I have specified below would give you bins of width 1 °C for temperature, ranging from -60 °C to + 55°C. And for dewpoint, you would get bins of width 0.5 °C, ranging from -80 °C to + 30 °C. There's no good reason I made smaller bins for dewpoint – just an example of the flexibility of the script. The value provided for `input_target_norm_file_name` is where I actually keep the normalization params on Hera.

```
python3 -u evaluate_model.py \
--input_prediction_dir_name="/home/ralager/fancy_new_model/predictions" \
--init_time_limit_strings "2021-01-01-00" "2022-12-31-23" \
```

---

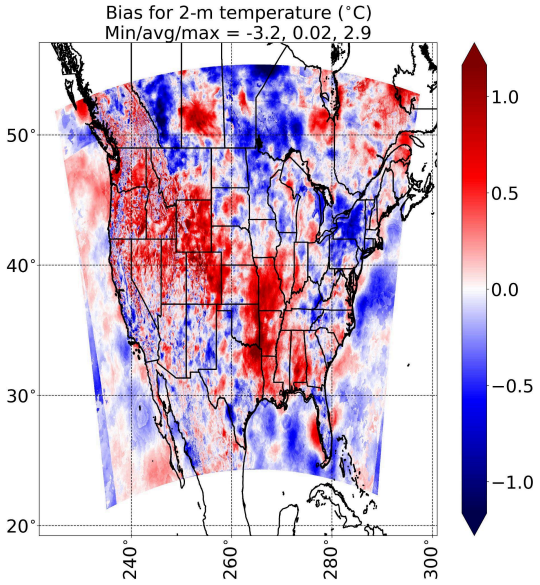
[3] The reliability diagram (or "reliability curve") plots two quantities against each other: the prediction (on the *x*-axis) and the conditional mean observation (on the *y*-axis). We call this the *conditional* mean observation because it is conditioned on the model's predicted value. Hence, the reliability curve tells you, for every predicted value of the target variable, what the expected observation is. For example, the reliability curve might tell you that, when the model predicts a temperature of 30 °C, the expected observed value is 29 °C – meaning that, for the predicted value of 30 °C, the model has a *conditional* bias of +1 °C. This could be true even if the model has a negative bias, or zero bias, overall. In practice, though, the reliability curve cannot contain one point for every possible infinitesimal predicted value: 30.000000 °C, 30.000001 °C, 30.000002 °C, etc. Thus, data samples must be *binned* according to the predicted value. For example, suppose that your target variable is temperature; you set the number of bins to 115; and you set the min and max bin edges to -60 °C and +55 °C. Then points in the reliability curve will correspond to bins [-60, -59) °C; [-59, -58) °C; …; [53, 54) °C; [54, 55] °C.

[4] I purposely didn't mention the Kolmogorov-Smirnov stuff above, because I'm not sure how useful it will be; I might delete this code.

```
--num_bootstrap_reps=1 \  # No bootstrapping in this case.
--target_field_names "temperature_2m_agl_kelvins" "dewpoint_2m_agl_kelvins" \
--input_target_norm_file_name="/scratch2/NCEPDEV/stmp/Ryan.Lagerquist/ml_for_national_blend_proje
ct/urma_data_final/processed/normalization_params_20170101-20211220.nc" \
--num_relia_bins_by_target 115 220 \
--min_relia_bin_edge_by_target -60 -80 \
--max_relia_bin_edge_by_target 55 30 \
--per_grid_cell=0 \  # Ungridded mode.
--keep_it_simple=0 \  # Compute everything.
--output_file_name_or_pattern="/home/ralager/fancy_new_model/predictions/ungridded_evaluation.nc"
```

2. plot_gridded_evaluation.py

After running `evaluate_model.py` in gridded mode, you can plot the results with this script. You will get plots that look like the following:



`plot_gridded_evaluation.py` has the following input arguments:

- `input_eval_file_name_or_pattern`: Path to evaluation file created by `evaluate_model.py`.
- `target_field_name`: Name of target field for which to plot evaluation metrics. This script deals with only one target field (but multiple metrics) at a time.
- `metric_names`: List of metrics you want to plot. If you type "`python3 plot_gridded_evaluation.py --help`" into a Bash terminal, the valid metric names will be listed for you.
- `min_colour_values` and `max_colour_values`: These arguments control the min/max value in the colour bar for each metric. Both `min_colour_values` and `max_colour_values` must be a list of length *M,* where *M* is the length of `metric_names`. If you don't want to specify the min and max plotting values directly (*e.g.,* because you

don't know what the min/max values should be yet), then use `min_colour_percentiles` and `max_colour_percentiles`, instead.

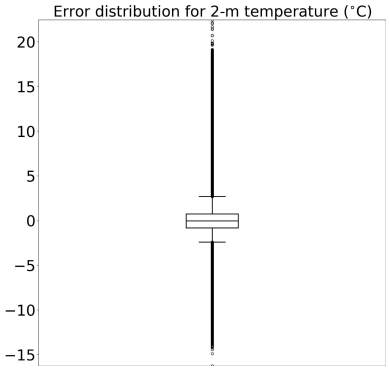
- `min_colour_percentiles` and `max_colour_percentiles`: These arguments may also control the min/max plotting value for each metric. Both `min_colour_percentiles` and `max_colour_percentiles` must be a list of length *M,* where *M* is the length of `metric_names`. For each metric, the given percentiles will be taken over all values in the grid, and these will become the min and max values in the colour bar.
- `output_dir_name`: Path to output directory. All the figures will be saved here as JPEG images.

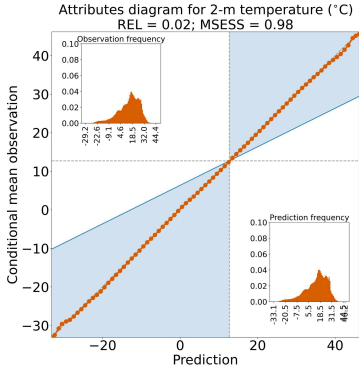
Here is an example of how you would call the script from the command line (or from a Bash script that you can submit as a Slurm job), assuming that you want the colour-bar limits to be the 0.5th and 99.5th percentile for every metric. That's what I did to create the above plot of temperature bias. Note that the min, max, and average value (again, spatial statistics taken over the grid) are always reported in the title.

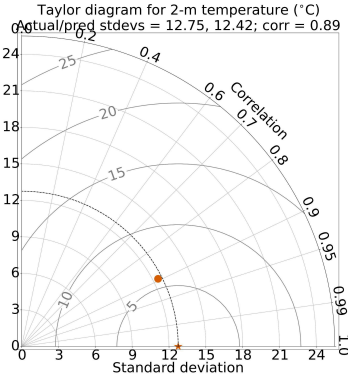
```
python3 -u plot_gridded_evaluation.py \
--input_eval_file_name_or_pattern="/home/ralager/fancy_new_model/predictions/gridded_evaluation.nc" \
--metric_names "target_standard_deviation" "prediction_standard_deviation" "target_mean" "prediction_mean"
"root_mean_squared_error" "mse_bias" "mse_variance" "mse_skill_score" "dual_weighted_mean_squared_error"
"dwmse_skill_score" "mean_absolute_error" "mae_skill_score" "bias" "correlation" "kling_gupta_efficiency"
"reliability" \
--min_colour_percentiles 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 \
--max_colour_percentiles 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 99.5 \
--output_dir_name="/home/ralager/fancy_new_model/predictions/gridded_evaluation_plots"
```

3. plot_ungridded_evaluation.py

After running `evaluate_model.py` in *un*gridded mode, you can plot the results with this script. For every target field, you will get a boxplot of errors, an attributes diagram, and a Taylor diagram. An example for each is shown below.







In the Taylor diagram, the grey contours marked with inline numbers (5, 10, 15, 20, 25) represent *centered* RMSE. There is a difference between typical RMSE and centered RMSE, as shown below:

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred},i} - y_{\text{obs},i})^2}$$

$$\text{Centered RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left[(y_{\text{pred},i} - \bar{y}_{\text{pred}}) - (y_{\text{obs},i} - \bar{y}_{\text{obs}})\right]^2}$$

The typical RMSE (first equation) includes both bias and random error. But in the centered RMSE, the mean prediction is removed from each prediction sample, and the mean observation is removed from each observation sample. This centering removes any effect produced by difference in the means between the two datasets (predictions and observations). What you're left with – the centered RMSE – compares only *variability* in the predictions relative to the observations. In other words, centered RMSE focuses on how well the shape/pattern of the predictions matches the observations, ignoring any difference in means.

`plot_ungridded_evaluation.py` has the following input arguments:

- `input_eval_file_name_or_pattern`: Path to evaluation file created by `evaluate_model.py`.
- `input_target_norm_file_name`: Path to normalization file for target variables. This file contains (among other things) the climo mean for every target variable, which is needed to plot the reference lines in the attributes diagram.
- `plot_full_error_distributions`: Boolean flag. If 1, the script will create boxplots; if 0, it will not. Creating the boxplots takes a little more time, but not too much – and I find them worthwhile.
- `confidence_level`: Confidence level for confidence intervals, ranging from 0 to 1. For example, 0.95 specifies a 95% confidence interval. This argument will be used only if the input file contains multiple bootstrap replicates, *i.e.,* if you ran `evaluate_model.py` with `num_bootstrap_reps` > 1. In this case, bootstrapping will be used to show a confidence interval in the attributes diagram.
- `report_metrics_in_titles`: Boolean flag. I suggest making this 1 all the time.
- `output_dir_name`: Path to output directory. Figures will be saved here as JPEG images.

Here is an example of how you would call `plot_ungridded_evaluation.py` from the command line or a Bash script.

```
python3 -u plot_ungridded_evaluation.py \
--input_eval_file_name_or_pattern="/home/ralager/fancy_new_model/predictions/ungridded_evaluation
.nc" \
--input_target_norm_file_name="/scratch2/NCEPDEV/stmp/Ryan.Lagerquist/ml_for_national_blend_proje
ct/urma_data_final/processed/normalization_params_20170101-20211220.nc" \
--plot_full_error_distributions=1 \
--confidence_level=0.95 \
--report_metrics_in_titles=1 \
--output_dir_name="/home/ralager/fancy_new_model/predictions/ungridded_evaluation_plots"
```

4. compute_spread_vs_skill.py

This script evaluates probabilistic forecasts, *i.e.,* the full ensemble instead of just the ensemble mean. This script computes all quantities needed to produce a spread-skill plot. The intention is that you run this script followed by `plot_spread_vs_skill.py`, where the latter creates the actual graphic. In other words, `compute_spread_vs_skill.py` computes the quantities needed to visualize a spread-skill plot but does not produce any graphics; then `plot_spread_vs_skill.py` produces the graphic. `compute_spread_vs_skill.py` has only an ungridded mode, which means that the spread-skill quantities are averaged over all space (the entire NBM grid) and averaged over the time period you specify in the input arguments. `compute_spread_vs_skill.py` does not have a gridded mode, for two reasons:

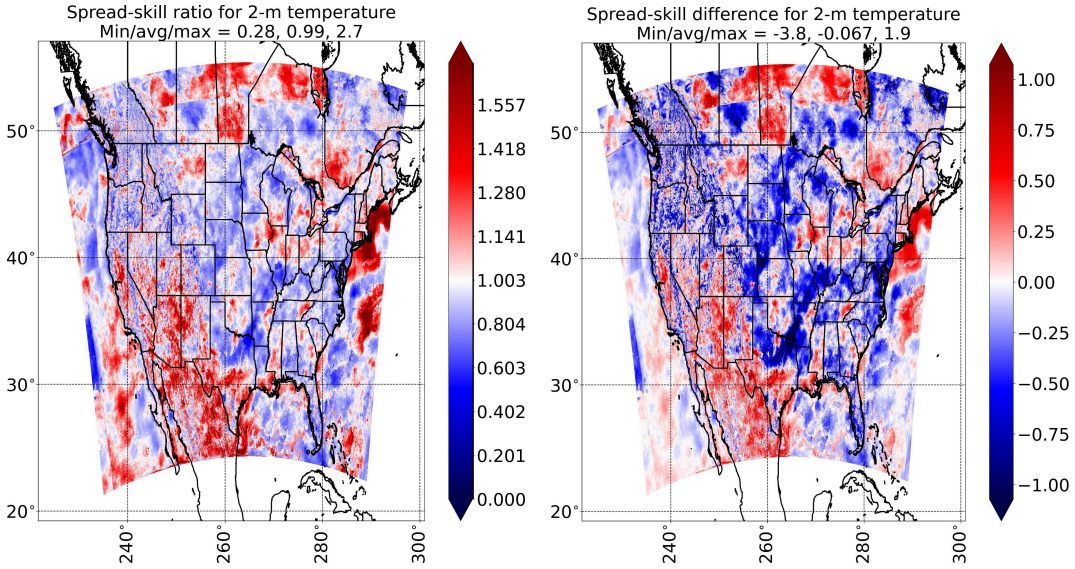
1. It would be *very* computationally expensive to produce a spread-skill plot for all ~3.7M grid points in the NBM grid.
2. No one would look at 3.7M spread-skill plots, anyways.

However, some useful summary statistics can be computed from the spread-skill plot, such as spread-skill reliability (SSREL) and spread-skill ratio (SSRAT). It occurred to me that it might be useful to visualize SSREL and SSRAT on a grid – just like `plot_gridded_evaluation.py` allows us to visualize bias, mean squared error, etc. on a grid. Furthermore, SSRAT can be computed without computing *everything* needed for the spread-skill plot, which makes computing just SSRAT much less computationally expensive than computing everything needed for the plot. Thus, `evaluate_model.py` has two extra input arguments that I didn't talk about:

- `compute_ssrat`: Boolean flag. If 0, `evaluate_model.py` won't bother computing SSRAT. If 1, `evaluate_model.py` will compute both SSRAT and spread-skill difference (SSDIFF).
- `compute_ssrel`: Boolean flag. If 0, `evaluate_model.py` won't bother computing SSREL. If 1, `evaluate_model.py` will compute SSREL.

If you run `evaluate_model.py` in gridded mode (with `per_grid_cell` = 1) and with `compute_ssrat` = 1, then you can plot maps of SSRAT and SSDIFF over the full NBM domain.

If you run `evaluate_model.py` in gridded mode and with `compute_ssrel` = 1, then you can plot maps of SSREL over the full NBM domain. Below is an example.



For more on the spread-skill plot, SSREL, and SSRAT, see Haynes *et al.* (2023). In general, I recommend Haynes *et al.* as a general tutorial paper on machine-learned uncertainty quantification, which is a huge part of this project. SSDIFF is similar to SSRAT but not included in Haynes *et al.*, so it is explained below for completeness.

$$\begin{cases} \text{SSRAT} = \dfrac{\overline{\text{SD}}}{\text{RMSE}} \\ \text{SSDIFF} = \overline{\text{SD}} - \text{RMSE} \end{cases}$$

On the right-hand side of either equation, the first value is the average ensemble standard deviation, and the second value is the root mean squared error (RMSE) incurred by the ensemble average.

`compute_spread_vs_skill.py` has the following input arguments:

- `input_prediction_dir_name`: Path to directory with prediction files. Every prediction file contains both predicted and actual (URMA) values, which are the main inputs needed for model evaluation. Within this directory, the expected file-naming structure is "predictions_<yyyy-mm-dd-HH>.nc", where the <yyyy-mm-dd-HH> part is the forecast-initialization time.

- `init_time_limit_strings`: This argument specifies the evaluation period, *i.e.,* the time period over which spread-skill quantities will be computed. For example, if your evaluation period is the years 2021 and 2022, init_time_limit_strings would be a list with ("2021-01-01-00", "2022-12-31-23"). Either way, the list must be two items long, with time stamps in the format I just gave.
- `target_field_names`: List of target fields to evaluate. The valid names here are "temperature_2m_agl_kelvins", "dewpoint_2m_agl_kelvins", "u_wind_10m_agl_m_s01", "v_wind_10m_agl_m_s01", and "wind_gust_10m_agl_m_s01".
- `num_bins_by_target`: Number of bins in spread-skill plot per target variable. This must be a list of length *T,* where *T* is the length of `target_field_names`.
- `min_bin_edge_by_target` and `max_bin_edge_by_target`: These arguments control the bin edges in the spread-skill plot for every target variable. Both `min_bin_edge_by_target` and `max_bin_edge_by_target` must be a list of length *T,* where *T* is the length of `target_field_names`. If you don't want to specify the min and max bin edges directly (*e.g.,* because you don't know what the min/max values should be yet), then use `min_bin_edge_prctile_by_target` and `max_bin_edge_prctile_by_target`, instead.
- `min_bin_edge_prctile_by_target` and `max_bin_edge_prctile_by_target`: These arguments can also be used to control bin edges. Each argument must be a list of length *T,* where *T* is the length of `target_field_names`. For example, suppose that the first item of `target_field_names` is "temperature_2m_agl_kelvins"; the first item of `min_bin_edge_prctile_by_target` is 0.5; and the first item of `max_bin_edge_prctile_by_target` is 99.5. This means that, in the spread-skill plot for temperature, the lowest bin edge will correspond to the 0.5^{th} percentile of all spread values, while the highest bin edge will correspond to the 99.5^{th} percentile of all spread values. There is one spread value per grid point per time step, and the "spread value" is specifically the ensemble standard deviation. In any case, I suggest defaulting these arguments to the 0^{th} and 100^{th} percentiles. If you would rather specify bin edges directly – and not by percentile – leave these arguments alone and use `min_bin_edge_by_target` and `max_bin_edge_by_target`.
- `output_file_name`: Path to output file. Spread-skill stuff will be saved here in NetCDF format.

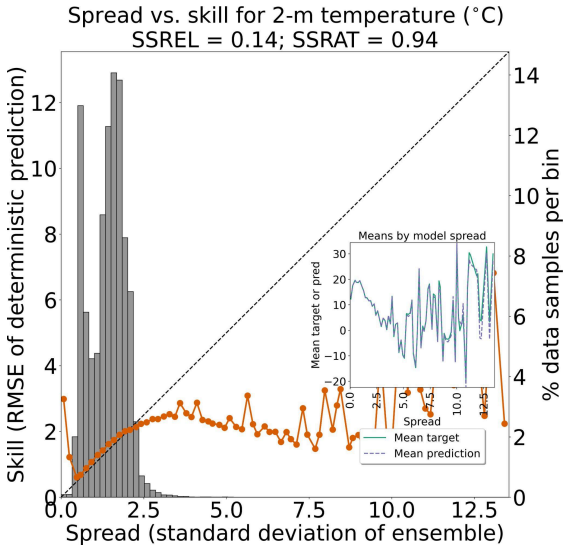
Here is an example of how you would call `compute_spread_vs_skill.py` from the command line or a Bash script. In this case, the spread-skill plot for temperature would end up with 100 bins, while that for dewpoint would end up with 50 bins. There is no good reason for this – just illustrating the flexibility of the script.

```
python3 -u compute_spread_vs_skill.py \
--input_prediction_dir_name="/home/ralager/fancy_new_model/predictions" \
```

```
--init_time_limit_strings "2021-01-01-00" "2022-12-31-23" \
--target_field_names "temperature_2m_agl_kelvins" "dewpoint_2m_agl_kelvins" \
--num_bins_by_target 100 50  \
--min_bin_edge_prctile_by_target 0 0 \
--max_bin_edge_prctile_by_target 100 100 \
--output_file_name="/home/ralager/fancy_new_model/predictions/spread_vs_skill.nc"
```

5. plot_spread_vs_skill.py

After running `compute_spread_vs_skill.py`, you can plot the results with this script. You will get one spread-skill plot for every target field. An example is shown below. For complete understanding of the spread-skill plot, I suggest reading Haynes *et al.* (2023).



`plot_spread_vs_skill.py` has very simple input arguments:

- `input_file_name`: Path to file with all the spread-skill quantities, created by `compute_spread_vs_skill.py`.
- `output_dir_name`: Path to output directory. Figures will be saved here as JPEG images.

6. make_pit_histograms.py

Like `compute_spread_vs_skill.py`, this script evaluates probabilistic forecasts, rather than deterministic forecasts. `make_pit_histograms.py` computes all quantities needed to plot a probability integral transform (PIT) histogram. If your probabilistic forecast takes the form of an ensemble – which is always true for my neural networks – then the PIT histogram is equivalent to a rank histogram. Like `compute_spread_vs_skill.py`, `make_pit_histograms.py` does not actually produce any graphics; this is done later by `plot_pit_histograms.py`. Also like `compute_spread_vs_skill.py`, `make_pit_histograms.py` has only an ungridded mode,

which means that it averages over all grid points. For an explanation of the PIT histogram, see Haynes *et al.* (2023).

`make_pit_histograms.py` has the following input arguments:

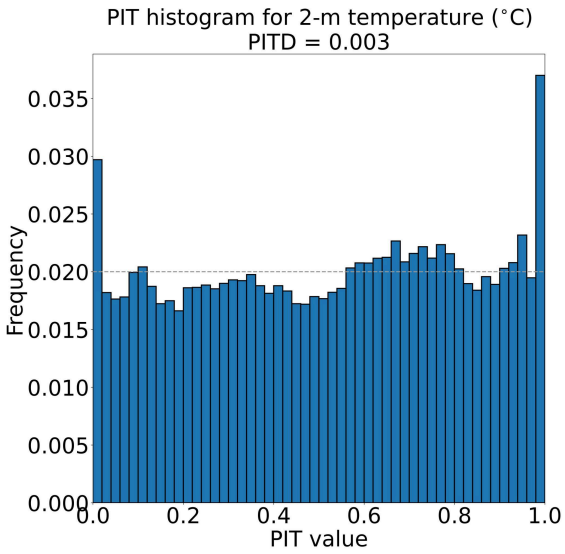
- `input_prediction_dir_name`: Same as input for `compute_spread_vs_skill.py`.
- `init_time_limit_strings`: Same as input for `compute_spread_vs_skill.py`.
- `target_field_names`: Same as input for `compute_spread_vs_skill.py`.
- `num_bins`: Number of bins in PIT histogram. This will be the same for every target field. I recommend making `num_bins = ensemble_size + 1`. In other words, if the neural net produces a 50-member ensemble, make `num_bins` = 51. This means that every possible PIT value (*i.e.,* every possible rank of the URMA observation within the ensemble) will get its own bin.
- `output_file_name`: Path to output file. PIT-histogram stuff will be saved here in NetCDF format.

Here is an example of how you would call `make_pit_histograms.py` from the command line or a Bash script. In this case, I'm assuming a 50-member ensemble, so specifying 51 bins.

```
python3 -u make_pit_histograms.py \
--input_prediction_dir_name="/home/ralager/fancy_new_model/predictions" \
--init_time_limit_strings "2021-01-01-00" "2022-12-31-23" \
--target_field_names "temperature_2m_agl_kelvins" "dewpoint_2m_agl_kelvins"
"u_wind_10m_agl_m_s01" "v_wind_10m_agl_m_s01" "wind_gust_10m_agl_m_s01" \
--num_bins=51 \
--output_file_name="/home/ralager/fancy_new_model/predictions/pit_histograms.nc"
```

7. plot_pit_histograms.py

After running `make_pit_histograms.py`, you can plot the results with this script. You will get one PIT histogram for every target field. An example is shown below.



`plot_pit_histograms.py` has very simple input arguments:

- `input_file_name`: Path to file with all the necessary quantities, created by `make_pit_histograms.py`.
- `output_dir_name`: Path to output directory. Figures will be saved here as JPEG images.

8. `run_discard_test.py`

The discard test is the other graphic I really like to use for evaluating probabilistic forecasts. `run_discard_test.py` computes all quantities needed to produce the discard plot, but it doesn't actually produce any graphics; this is done later by `plot_discard_test.py`. `run_discard_test.py` has only an ungridded mode, which means that it aggregates over all grid points. For an explanation of the discard test, see Haynes *et al.* (2023).

`run_discard_test.py` has the following input arguments:

- `input_prediction_dir_name`: Same as input for `compute_spread_vs_skill.py`.
- `init_time_limit_strings`: Same as input for `compute_spread_vs_skill.py`.
- `target_field_names`: Same as input for `compute_spread_vs_skill.py`.
- `discard_fractions`: List of discard fractions to include in test, all ranging from (0, 1) non-inclusive. A discard fraction of 0.0 ("no cases thrown out") will be included in the test by default, so you don't need to specify it. My default list of discard fractions is (0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70 0.75 0.80 0.85 0.90 0.95). This means that – during the course of the discard test – model error (MSE of the ensemble mean) will be evaluated with the 5% highest-uncertainty cases thrown out, the 10% highest-uncertainty cases thrown out, …, up to the 95% highest-uncertainty cases thrown out.

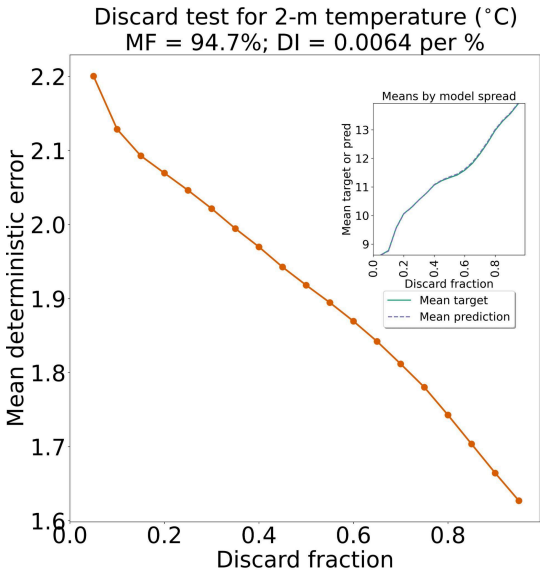
- `output_file_name`: Path to output file. Discard-test stuff will be saved here in NetCDF format.

Here is an example of how you would call `run_discard_test.py` from the command line or a Bash script.

```
python3 -u run_discard_test.py \
--input_prediction_dir_name="/home/ralager/fancy_new_model/predictions" \
--init_time_limit_strings "2021-01-01-00" "2022-12-31-23" \
--target_field_names "temperature_2m_agl_kelvins" "dewpoint_2m_agl_kelvins"
"u_wind_10m_agl_m_s01" "v_wind_10m_agl_m_s01" "wind_gust_10m_agl_m_s01" \
--discard_fractions 0.05 0.10 0.15 0.20 0.25 0.30 0.35 0.40 0.45 0.50 0.55 0.60 0.65 0.70 0.75
0.80 0.85 0.90 0.95 \
--output_file_name="/home/ralager/fancy_new_model/predictions/discard_test.nc"
```

9. plot_discard_test.py

After running `run_discard_test.py`, you can plot the results with this script. You will get one graph for every target field. An example is shown below.



`plot_discard_test.py` has very simple input arguments:

- `input_file_name`: Path to file with all the necessary quantities, created by run_discard_test.py.
- `output_dir_name`: Path to output directory. Figures will be saved here as JPEG images.

10. plot_error_distributions.py

This script plots error distributions, with a special focus on **extreme values**. Unlike the other plotting scripts (`plot_gridded_evaluation.py`, `plot_spread_vs_skill.py`, etc.), you don't need to run a separate number-crunching script (`evaluate_model.py`, `compute_spread_vs_skill.py`, etc.) before this plotting script. In other words, `plot_error_distributions.py` does the number-crunching *and* plotting.
`plot_error_distributions.py` creates five plots, described below, for every target variable. Note that all of these plots evaluate only the ensemble-mean prediction, with no regard to ensemble spread or anything else about the full ensemble.

i. **The error distribution (prediction minus target) as a function of the target value.** Atomic data samples[5] are binned by target value, and for each bin, the error distribution is plotted as either a violin plot or a boxplot. All violin plots or boxplots are presented on the same axes, so that you can see, at a glance, how the error distribution changes with the target value.

ii. **The error distribution (prediction minus target) as a function of the predicted value.** Thus, plot #ii is the same as plot #i, except that atomic data samples are binned by predicted value instead of target value. Plot #i shows how the error distribution changes with the target value; plot #ii shows how it changes with the predicted value. With both plot types #i and #ii, it's often especially interesting to see what happens at the *extreme* values, *i.e.,* the left and right ends of the *x*-axis.

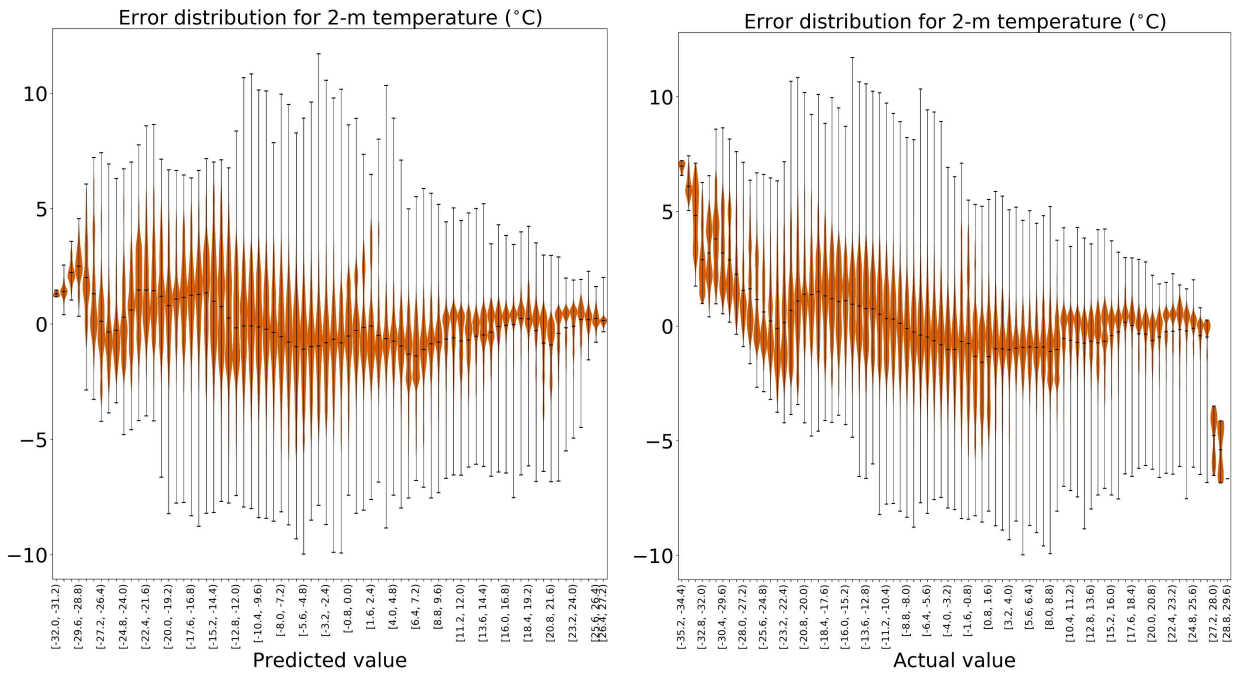
iii. **The empirical probability-density function (PDF) of both distributions (the target values and predicted values).** The empirical PDF is created via kernel-density estimation (KDE). This plot shows you, at a glance, which values of the target variable are being predicted more/less often than they should.

iv. **The left tail of each empirical PDF (one for the predicted values, one for the target values).** This is the same as plot #iii, except that it's zoomed into the left tail of the distribution. This plot shows you, at a glance, for extremely low values of the given variable, which values the model is predicting more/less often than it should.

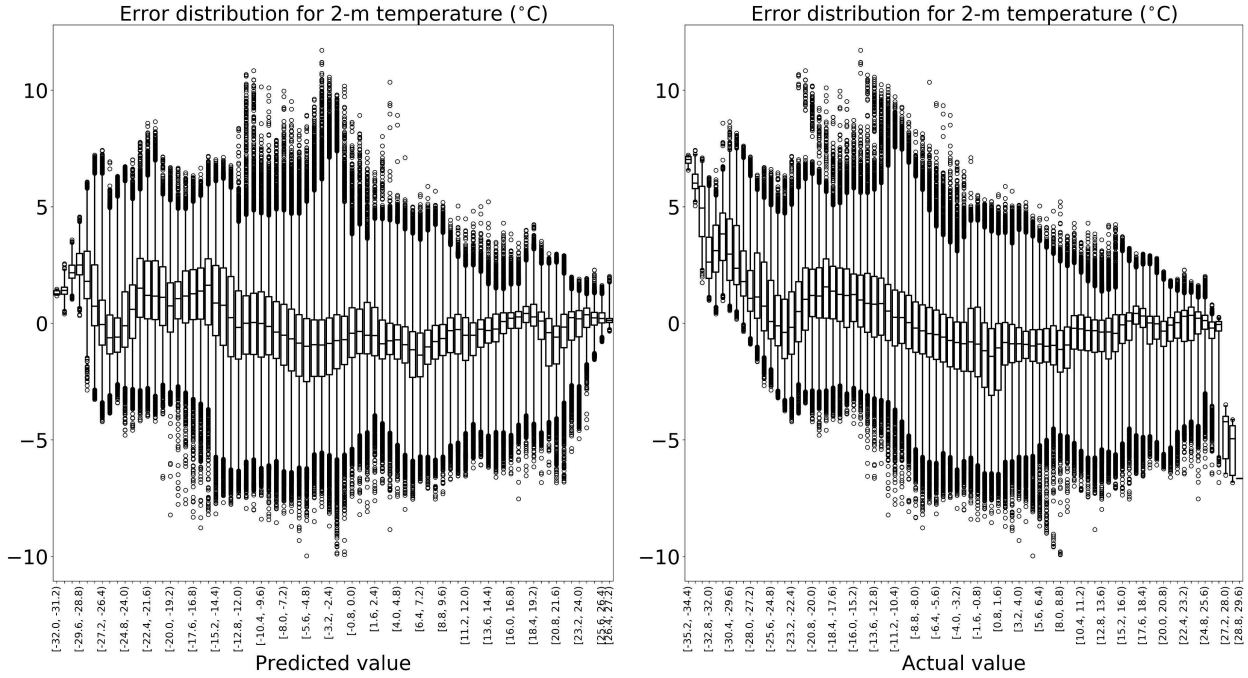
v. **The right tail of each empirical PDF.** This is the same as plot #iv, except zoomed into the right tail instead of the left tail. Hence, this plot focuses on extremely high values, rather than extremely low values. For both plots #iv and #v, like plot #iii, the ideal situation is that the two PDFs are identical – *i.e.,* that the model matches the true data distribution.

Below are examples of plots #i and #ii, using the violin option (instead of the boxplot option).

[5] One atomic data sample = one grid point at one valid time.



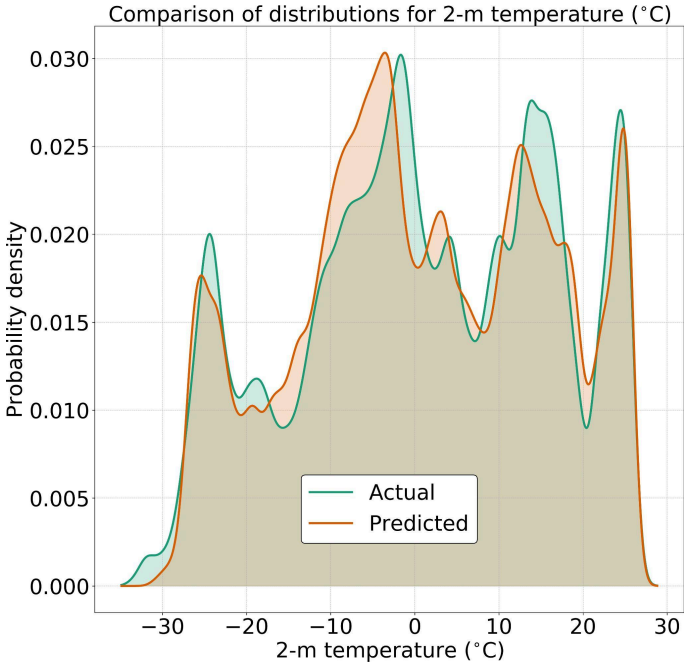
Below are examples of plots #i and #ii, using the boxplot option. Note that both sets of plots (the two violin plots and the two boxplots) are showing the exact same data and therefore have the same overall shapes – they're just presenting the same data in different formats.



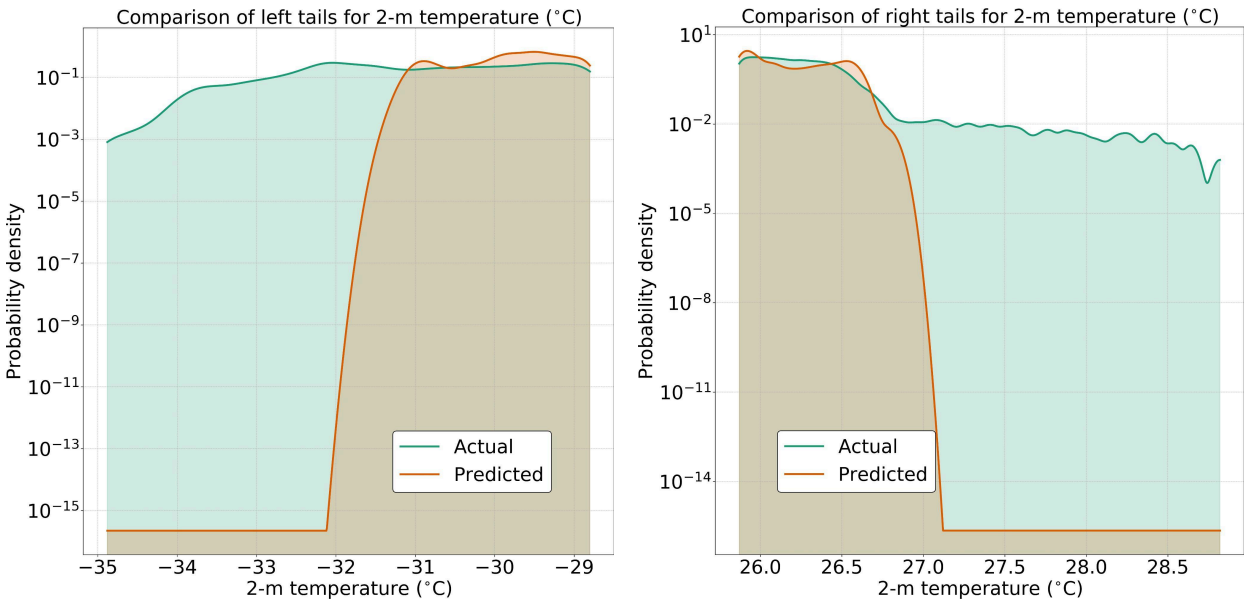
Based on the violin/box plots above, you can make the following inferences, for example:

- In extreme cold situations (when the actual temperature is < -30 °C), the model has a large positive bias. The entire error distribution is above 0 °C (meaning that the model *always* overpredicts), and the mean error reaches +7 °C for the lowest bin (actual temperature from -35.2 to -34.4 °C). In other words, the model has a bias of +7 °C for the coldest actual temperatures.
- In extreme hot situations (when the actual temperature is above 27 °C), the model has a large negative bias. The entire error distribution is below 0 °C (meaning that the model *always* underpredicts), and the mean error reaches -6 °C for the highest bin (actual temperature from 28.8 to 29.6 °C). In other words, the model has a bias of -6 °C for the warmest actual temperatures.
- Combining the two conclusions above: the model underdoes extreme values. When the actual temperature is extremely cold/warm, you can expect the model predictions to *not be extreme enough*.

Below is an example of plot #iii:



Below are examples of plots #iv and #v. Based on these plots, you can make the same conclusions (about the model undergoing extremes) as were made from plots #i and #ii. Note that both plots have a logarithmic *y*-axis, allowing you to *really* zoom in on the extreme values and compare the shapes of the two distributions.



At long last, here is a description of the input arguments for `plot_error_distributions.py`:

- `input_prediction_dir_name`: Same as input for `compute_spread_vs_skill.py`.
- `init_time_limit_strings`: Same as input for `compute_spread_vs_skill.py`.
- `evaluate_month`: Integer from 1 to 12. The default value is -1, in which case the plots will be based on predictions from all months. But if this value is from 1…12, only predictions *valid in* the given month will be used to make the plots. For example, if your time period (specified by `init_time_limit_strings`) is 2021-2022 and you set `evaluate_month` = 3, the plots will be based on forecasts valid in March 2021 and March 2022.
- `evaluate_hour`: Integer from 0 to 23. The default value is -1, in which case the plots will be based on predictions from all UTC hours. But if this value is from 0…23, only predictions *valid at* the given UTC hour will be used to make the plots. For example, if your time period (specified by `init_time_limit_strings`) is 2021-2022 and you set `evaluate_hour` = 18, the plots will be based on forecasts valid at 18Z from all days in 2021 and 2022.
- `target_field_names`: Same as input for `compute_spread_vs_skill.py`.
- `num_bins_by_target`: Number of bins in violin/box plot per target variable. This must be a list of length *T,* where *T* is the length of `target_field_names`.
- `min_bin_edge_by_target` and `max_bin_edge_by_target`: These arguments control the bin edges in the violin/box plot for every target variable. Both `min_bin_edge_by_target` and `max_bin_edge_by_target` must be a list of length *T,* where *T* is the length of `target_field_names`.

- `violin_or_box_plots`: Boolean flag. If you want plots #i and #ii to be formatted as violin plots, make this 1. Otherwise, make this 0.
- `left_tail_percentile`: Percentile used to define the left tail of the distribution, for the purpose of making plot #iv. For example, if you want the "left tail" to be the bottom 0.5% of values, make this 0.5.
- `right_tail_percentile`: Percentile used to define the right tail of the distribution, for the purpose of making plot #v. For example, if you want the "right tail" to be the top 1.0% of values, make this 1.0.
- `max_num_pdf_values`: Maximum number of values used to make the PDFs for plot #iii. This is the maximum number of atomic data samples (defined above). If you want to use *all* values, just make this argument -1. But if you have a huge amount of data, I suggest actually using `max_num_pdf_values`; a loose suggestion is 50 million. In that case, 50 million atomic data samples will be randomly subset from the full list of atomic data samples, and only these will be used for the PDF in plot #iii. The reason for this argument is that KDE can take a very long time, especially given the size of the NBM grid (~4M pixels). For example, if the script is reading in forecasts from 500 time steps, you then have 500 x ~4M = ~2 billion atomic data samples, which is *a lot* for KDE. It will take well over an hour to generate plot #iii *for each variable*.
- `output_dir_name`: Path to output directory, where all the figures will be saved.

Here is an example of how you would call `plot_error_distributions.py` from the command line or a Bash script. In this case, the violin plot for temperature would end up with 100 bins, while that for dewpoint would end up with 50 bins. There is no good reason for this – just illustrating the flexibility of the script. Also, note that bin edges are in units of the target variable (actually °C here, because the "`kelvins`" in both "`temperature_2m_agl_kelvins`" and "`dewpoint_2m_agl_kelvins`" is a dirty lie). Hence, bins for dewpoint will range from -80 to +35 °C, while bins for temperature will range from -60 to +55 °C. Any bin with zero cases (*i.e.,* zero atomic data samples) will not be plotted, meaning that there just won't be a violin/box plot for that bin.

```
python3 -u plot_error_distributions.py \
--input_prediction_dir_name="/home/ralager/fancy_new_model/predictions" \
--init_time_limit_strings "2021-01-01-00" "2022-12-31-23" \
--target_field_names "temperature_2m_agl_kelvins" "dewpoint_2m_agl_kelvins" \
--num_bins_by_target 100 50  \
--min_bin_edge_by_target -80 -60 \
--max_bin_edge_by_target 35 55 \
--violin_or_box_plots=1 \  # Use violin plots.
--left_tail_percentile=0.5 \
--right_tail_percentile=1.0 \
--max_num_pdf_values=50000000 \
--output_file_name="/home/ralager/fancy_new_model/predictions/error_distribution_plots"
```

Part II: Temporal stratification

Temporally stratified model evaluation is handled by four different Python scripts. These are described in turn.

1. `evaluate_model.py`

As discussed already in Part I, this script has the following properties:

- Evaluates deterministic forecasts (not uncertainty estimates), so cares only about the ensemble mean
- Can run in gridded mode or ungridded mode. In gridded mode, evaluation metrics are stratified by NBM grid point. In ungridded mode, evaluation metrics are not stratified spatially at all.
- Computes a long list of metrics for every target field (see long list in Part I)
- For every target field, also computes everything needed to plot the attributes diagram and Taylor diagram

However, in Part I, I obfuscated one property of this script: it can be used to stratify temporally (by hour-of-day or month-of-year). Temporal stratification is controlled by two input arguments:

- `evaluate_month`: Integer from 1 to 12. The default value is -1, in which case evaluation will *not* be stratified by month. But if you *do* want to stratify by month, make this an integer from 1 to 12, indicating the month for which you want to compute evaluation metrics. For example, if your time period (specified by `init_time_limit_strings`) is 2021-2022 and you set `evaluate_month` = 3, the script will compute evaluation metrics only for forecasts *valid in* (not initialized in) March within the given time period. In other words, evaluation metrics will be based on all forecasts valid in March 2021 and March 2022.
- `evaluate_hour`: Integer from 0 to 23. The default value is -1, in which case evaluation will *not* be stratified by hour. But if you *do* want to stratify by hour (specifically UTC hour), make this an integer from 0 to 23, indicating the hour you care about. For example, if your time period (specified by `init_time_limit_strings`) is 2021-2022 and you set `evaluate_hour` = 6, the script will compute evaluation metrics only for forecasts *valid at* (not initialized at) 0600 UTC within the given time period. In other words, evaluation metrics will be based on all forecasts valid at 0600 UTC in 2021-2022.

To compute evaluation metrics for every month of the year, you would need to run `evaluate_model.py` 12 times, each time with a different argument for `evaluate_month`. To compute evaluation metrics for every UTC hour of the day, you would need to run

`evaluate_model.py` 24 times, each time with a different argument for `evaluate_hour`. All input arguments other than `evaluate_month` and `evaluate_hour` behave as described in Part I, with one exception:

- `output_file_name_or_pattern`: As the variable name indicates, this can be either a direct path to the output file or a *pattern* for the path to the output file. If you're running `evaluate_model.py` without temporal stratification – *i.e.,* with both `evaluate_month` = -1 and `evaluate_hour` = -1 – this argument will be a direct path. But if you're running `evaluate_model.py` with temporal stratification, this argument will be only *part* of the path to the output file. For example, if this argument is "/home/ralager/metrics.nc" and you set `evaluate_hour` = 6, the actual file name will be "/home/ralager/metrics_hour06.nc". If this argument is "/home/ralager/metrics.nc" and you set `evaluate_month` = 3, the actual file name will be "/home/ralager/metrics_month03.nc".

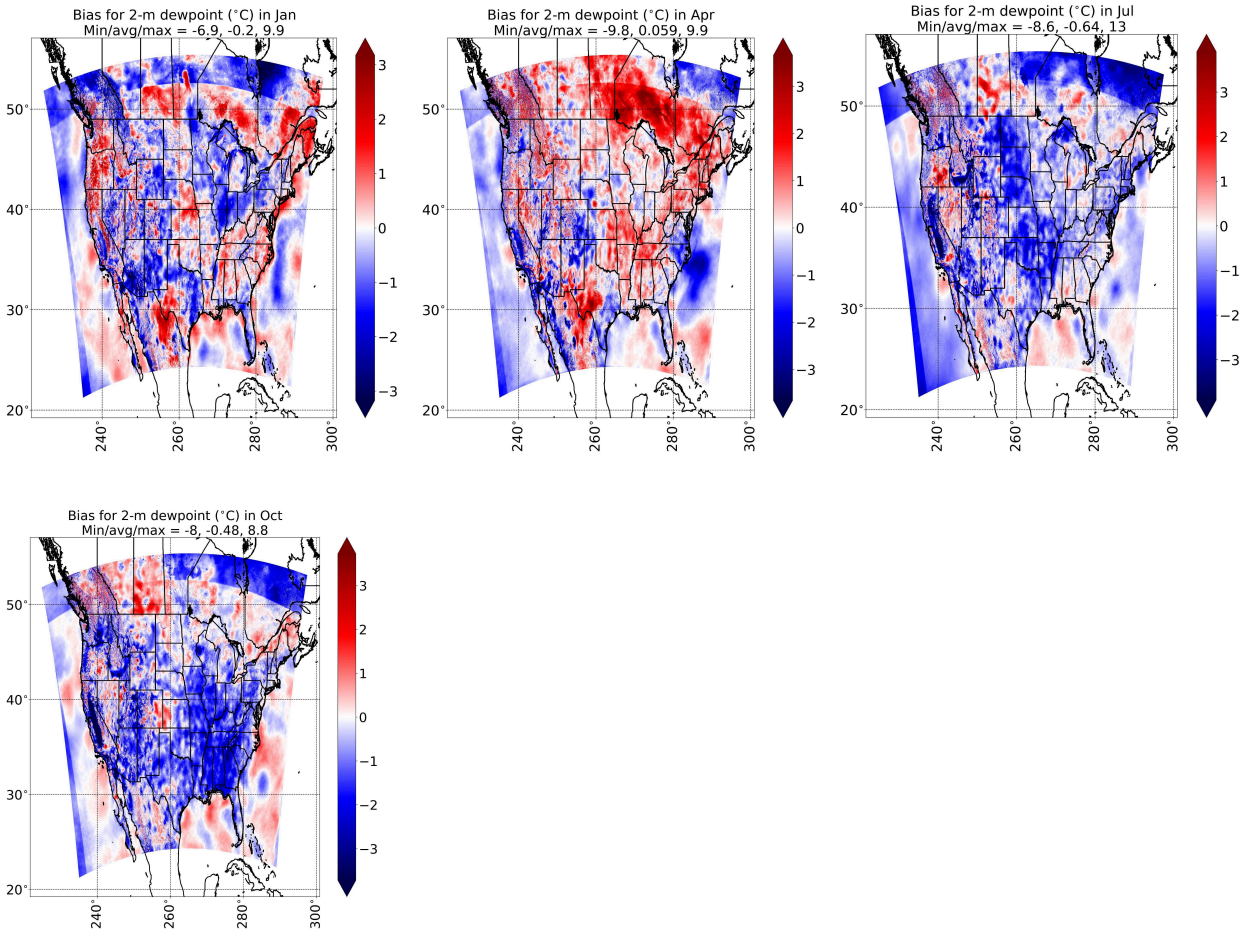
2. `plot_gridded_evaluation.py`

As discussed already in Part I, this script plots results from the gridded mode of `evaluate_model.py`. If you ran `evaluate_model.py` *without* temporal stratification, then you will run `plot_gridded_evaluation.py` as specified in Part I. But if you ran `evaluate_model.py` *with* temporal stratification, then you will run `plot_gridded_evaluation.py` differently. Three input arguments will change:

- `input_eval_file_name_or_pattern`: As the variable name indicates, this can be either a direct path to the input file or a *pattern* for the path to the input files. This argument should be set the same as `output_file_name_or_pattern` when you ran `evaluate_model.py`. So if you set `output_file_name_or_pattern` to "/home/ralager/metrics.py", then you should set `input_eval_file_name_or_pattern` here to "/home/ralager/metrics.py" – regardless of what the next two input arguments look like.
- `by_month`: Boolean flag. If 0, `plot_gridded_evaluation.py` will *not* look for files stratified by month. But if 1, then `plot_gridded_evaluation.py` *will* look for files stratified by month – and will create one set of plots for every month. For example, if `input_eval_file_name_or_pattern` = "/home/ralager/metrics.py" and `by_month` = 1, this script will look for files "/home/ralager/metrics_month01.py", "/home/ralager/metrics_month02.py", …, "/home/ralager/metrics_month12.py".
- `by_hour`: Boolean flag. If 0, `plot_gridded_evaluation.py` will *not* look for files stratified by UTC hour. But if 1, then `plot_gridded_evaluation.py` *will* look for files stratified by UTC hour – and will create one set of plots for every hour of the day. For example, if `input_eval_file_name_or_pattern` = "/home/ralager/metrics.py" and

by_hour = 1, this script will look for files "/home/ralager/metrics_hour00.py", "/home/ralager/metrics_hour01.py", …, "/home/ralager/metrics_hour23.py".

For example, in monthly mode (with by_month = 1), instead of getting one bias plot for every target variable, you will get one for every {target variable, month} combo. Below are some examples for monthly dewpoint.

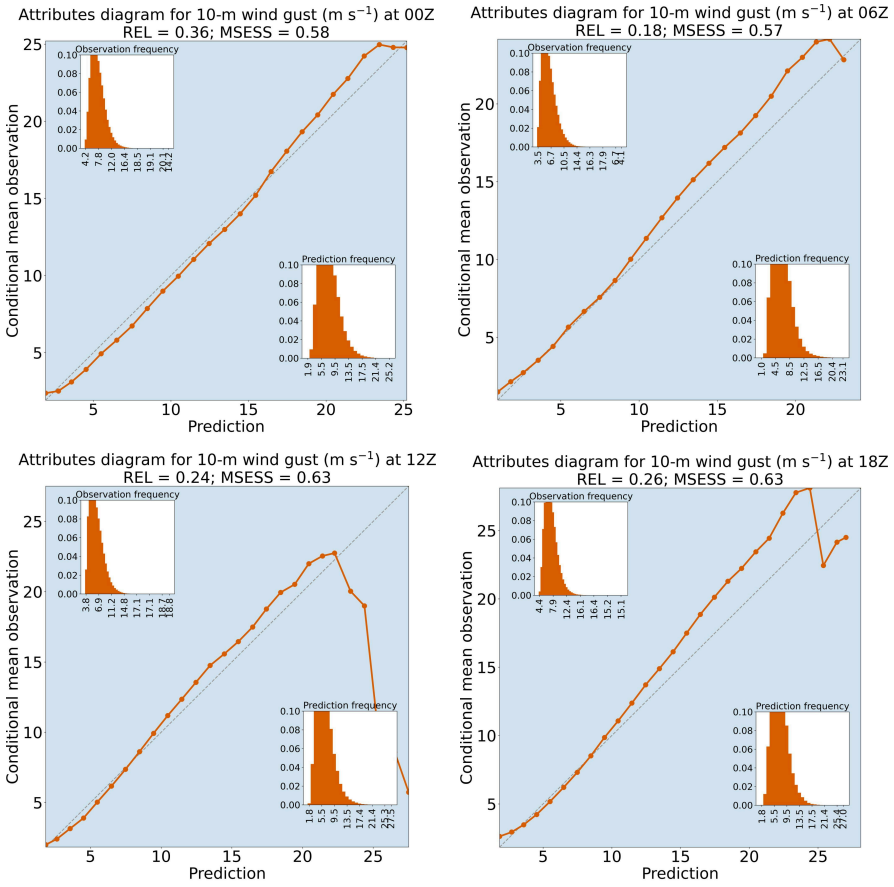


3. plot_ungridded_evaluation.py

As discussed already in Part I, this script plots results from the ungridded mode of evaluate_model.py. If you ran evaluate_model.py *without* temporal stratification, then you will run plot_ungridded_evaluation.py as specified in Part I. But if you ran evaluate_model.py *with* temporal stratification, then you will run plot_ungridded_evaluation.py differently. Three input arguments will change:

- `input_eval_file_name_or_pattern`: See documentation in Section II-2 for `plot_gridded_evaluation.py`.
- `by_month`: See documentation in Section II-2 for `plot_gridded_evaluation.py`.
- `by_hour`: See documentation in Section II-2 for `plot_gridded_evaluation.py`.

For example, in hourly mode (with `by_hour` = 1), instead of getting one attributes diagram for every target variable, you will get one for every {target variable, hour} combo. Below are some examples for hourly wind gust.



4. plot_eval_by_hour_or_month.py

This script plots ungridded (*i.e.,* spatially aggregated – *i.e.,* not spatially stratified) evaluation metrics on a line graph, where the *x*-axis is either hour or month. The graphs created by this script allow you to quickly see ("at a glance") how model performance varies by time-of-day or month-of-year. The input arguments are as follows:

- `input_eval_file_pattern`: This is a pattern for the path to the input files. For example, if this argument is "/home/ralager/metrics.nc" and you set `by_hour` = 1, the actual file names sought will be "/home/ralager/metrics_hour00.nc",

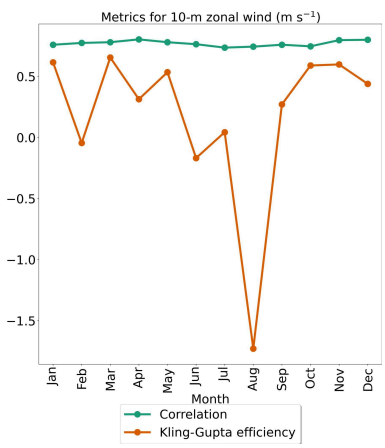
"/home/ralager/metrics_hour01.nc", …, "/home/ralager/metrics_hour23.nc". If this argument is "/home/ralager/metrics.nc" and you set `by_month` = 1, the actual file names sought will be "/home/ralager/metrics_month01.nc", "/home/ralager/metrics_month02.nc", …, "/home/ralager/metrics_month12.nc".

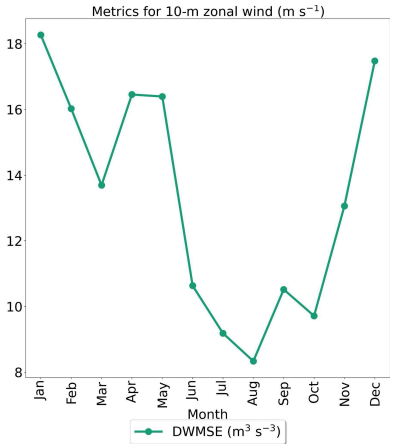
- `by_hour`: Boolean flag. If 1, the graphs created will plot evaluation metrics by hour. Only one of `by_hour` and `by_month` should be set to 1; the script cannot do both hourly and monthly in the same call.
- `by_month`: Boolean flag. If 1, the graphs created will plot evaluation metrics by month.
- `target_field_names`: List of target fields for which to plot graphs. The valid names here are "temperature_2m_agl_kelvins", "dewpoint_2m_agl_kelvins", "u_wind_10m_agl_m_s01", "v_wind_10m_agl_m_s01", and "wind_gust_10m_agl_m_s01". The script will plot one graph for every combination of {target field, metric}.
- `metric_names`: List of metrics for which to plot graphs. Again, the script will plot one graph for every combination of {target field, metric}. Valid options are keys in the dictionary `METRIC_DICT_TO_VERBOSE` defined at the top of `plot_eval_by_hour_or_month.py`. To be verbose, I will repeat the list of valid options here:
  - `'target_standard_deviation'` (stdev of actual values)
  - `'prediction_standard_deviation'` (stdev of predictions)
  - `'target_mean'` (mean of actual values)
  - `'prediction_mean'` (mean of predictions)
  - `'root_mean_squared_error'`
  - `'mse_bias'` (bias component of MSE)
  - `'mse_variance'` (variance component of MSE)
  - `'mse_skill_score'`
  - `'dual_weighted_mean_squared_error'`
  - `'dwmse_skill_score'`
  - `'kolmogorov_smirnov_statistic'`
  - `'kolmogorov_smirnov_p_value'`
  - `'mean_absolute_error'`
  - `'mae_skill_score'`
  - `'bias'` (mean value of [predicted minus actual])
  - `'correlation'` (Pearson correlation)
  - `'kling_gupta_efficiency'`
  - `'reliability'`
  - `'spread_skill_ratio'`
  - `'spread_skill_difference'`
  - `'spread_skill_reliability'`
  - `'spatial_min_bias'`

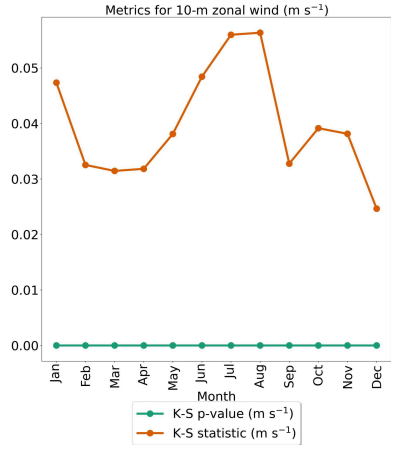
- `'spatial_max_bias'`

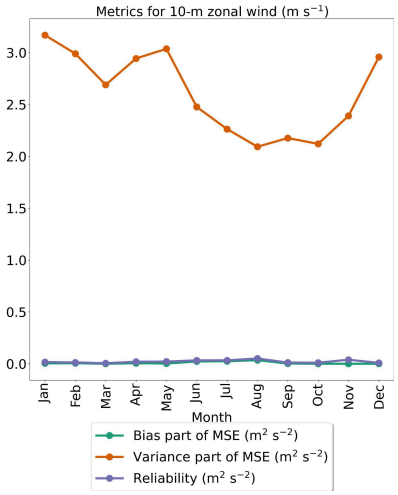
- `output_dir_name`: Path to output directory. All the graphs will be saved here.

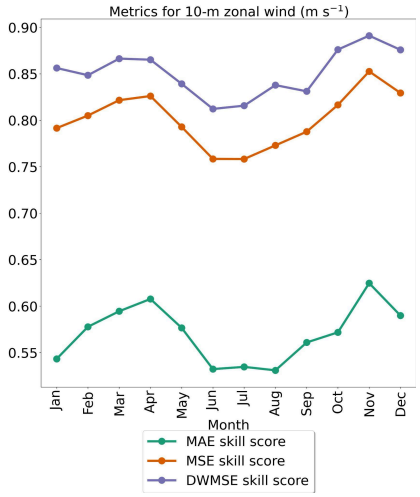
Below are some examples for monthly *u*-wind.

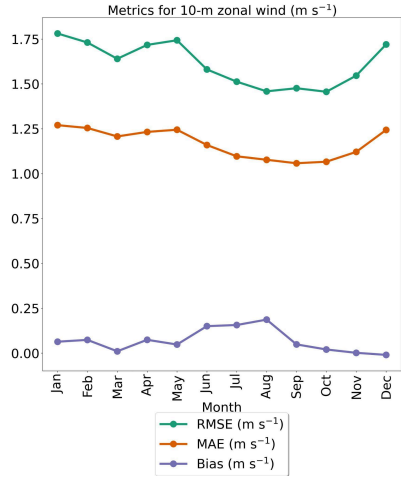


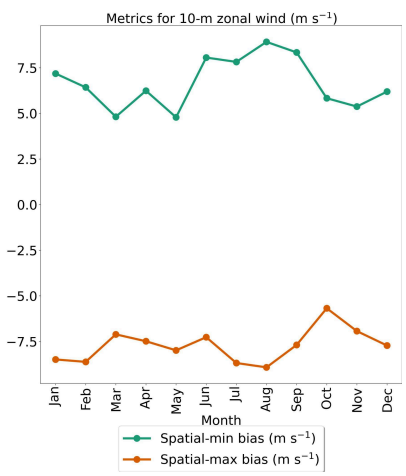


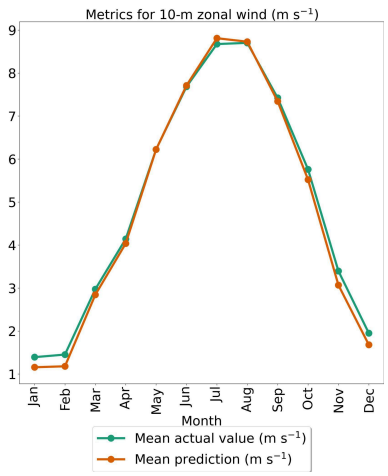


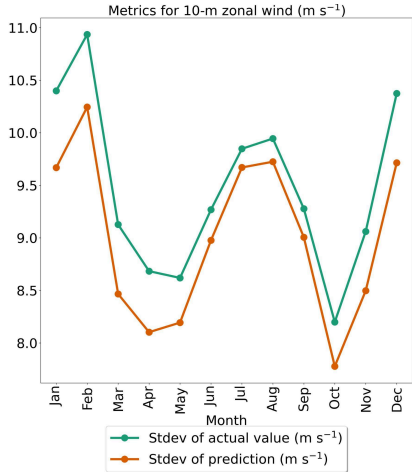







Metrics for 10-m zonal wind (m s^{-1})
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month
Spatial-min bias (m s^{-1})
Spatial-max bias (m s^{-1})


Metrics for 10-m zonal wind (m s^{-1})
9
8
7
6
5
4
3
2
1
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month
Mean actual value (m s^{-1})
Mean prediction (m s^{-1})


Metrics for 10-m zonal wind (m s^{-1})
11.0
10.5
10.0
9.5
9.0
8.5
8.0
Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
Month
Stdev of actual value (m s^{-1})
Stdev of prediction (m s^{-1})