EPA NMF-PY Algorithm Details

Summary

The details provided in this notebook are focused on describing two algorithms which are capable of meeting the constraints of PMF5, produce models which have a high correlation with the outputs of PMF5, and are able to generate models with a loss value that matches or is lower than PMF5. The algorithm used in PMF5, Multi-Linear Engine 2 (ME-2), was built upon the original ME-1 algorithm that is detailed in the publication by Pentti Paatero *The Multilinear Engine: A Table-Driven, Least Squares Program for Solving Multilinear Problems, including the n-Way Parallel Factor Analysis Mode*. The algorithm uses a modified conjugate gradient method using projection to maintain non-negativity of the factor profile matrix. The algorithm falls under the category of a semi-NMF algorithm, as not all the matrices are required to be positive. The reason for not fully enforcing non-negativity on the matrices is to help prevent a higher positive value bias due to having uncertainty and small values in the input datasets. Another reason was to allow datasets which may have small negative values in the inputs because of how the data was collected, where the input data represents a distribution rather than a single data point.

Algorithm Requirements

The constraints and requirements of the PMF5 algorithm which had to be considered in the new NMF-PY algorithms are:

- The loss function had to remain the same as the ME2 loss function.
- The output of NMF-PY must be able to produce results which have a high correlation with the ME2 output. Reproducing the output of ME2 exactly is highly improbable due to any differences in the update procedure and the randomness of the starting state.
  - A correlation of greater than 0.9 averaged across the factor profile, factor contributions and the concentration output.
  - A correlation of greater than 0.9 for the factor profile.
- The algorithm needs to properly function when negative values are present in the input data, and allow for negative values to be present in the factor contribution matrix.
- The output model needs to have a loss value that is comparable to PMF5.
- The algorithm should be as fast or faster than ME2.

Loss Function

The loss function that is used in ME2, and is described in the PMF5 User's Manual is defined as:

$$Q = \sum_{i=1}^{n}\sum_{j=1}^{m}\left[\frac{V_{ij} - \sum_{k=1}^{K} W_{ik}H_{kj}}{U_{ij}}\right]^2$$

here V is the input data matrix of features (columns=M) by samples (rows=N), U is the uncertainty matrix of the input data matrix, W is the factor contribution matrix of samples by factors=k, H is the factor profile of factors by features.

Convergence

These algorithms all use a similar convergence criteria as the stopping condition. PMF5 has a tiered approach which is described full in the User's Manual. The two NMF-PY algorithms offers two parameters which can be used to adjust the convergence criteria and allow for tuning of the final output. The two parameters are *converge_delta* and *converge_n*, and is simply the number of steps, *converge_n* where the change in Q is less than *converge_delta* the model is considered converged and updates stop. These values in testing are typically set to *converge_delta* = 0.1 and *converge_n* = 10 (these values were chosen to speed up model convergence during testing and development).

Initial Conditions

The performance of NMF algorithms is very sensitive to the initial conditions of the model, in this case the choices for the W and H matrices. There are multiple methods for initialization that are typically used for NMF. The method used by PMF5/ME2 is unknown. For NMF-PY, we have provided three different methods of initialization:

- Random sampling from a normal distribution using the square root of the mean values of the input dataset V by row N for W and by column M for H. This is the default method in most NMF packages.
- K-Means clustering, where the input dataset is normalized (can also be clustered without normalization) and k clusters calculated. Allocation of a factor to a cluster is set to 1.0 and all other values are equal to $\frac{1}{k}$.
- C-Means clustering, also known as fuzzy k-means clustering. Which is similar to K-Means but assignment to a cluster is not a binary value but continuous as calculated by the distance to the cluster centroids and the ratio of those distances.

Other methods of initialization exist but these are what we are currently providing in NMF-PY.

NMF-PY Algorithms

We have implemented two algorithms, one of which fully satisfies the algorithm requirements stated above and another which satisfies all by the condition of allowing for negative values.

LS-NMF

The first algorithm option we provide is a well documented algorithm called *LS-NMF*, least-squares nmf, and is available in the R NMF package. The ls-nmf algorithm is documented in *LS-NMF: A modified non-negative matrix factorization algorithm utilizing uncertainty estimates* and the R NMF package can be found at https://cran.r-project.org/web/packages/NMF/index.html. The NMF-PY versions of these algorithms converts the uncertainty of U into weights defined as $Uw = \frac{1}{U^2}$. The update equations then become:

$$H_{t+1} = H_t \circ \frac{W_t(V \circ Uw)}{W_t((W_t H_t) \circ Uw)}$$

$$W_{t+1} = W_t \circ \frac{(V \circ Uw)H_{t+1}}{((W_t H_{t+1}) \circ Uw)H_{t+1}}$$

The ls-nmf algorithm requires that all matrices be non-negative but is able to produce models with a lower loss value than PMF5 and is significantly faster than ME2. We include this algorithm as an option for when full non-negativity is permitted in the models because of the algorithm's performance and efficiency.

Weighted Semi-NMF

The weighted semi-nmf (ws-nmf) algorithm is a more complicated algorithm that more closely resembles the ME1 algorithm. The ws-nmf algorithm satisfies all of the requirements for a replacement algorithm of PMF5. The implemented algorithm was developed utilizing elements from two different publications, neither of which provided the complete algorithm we required for NMF-PY. *Semi-NMF and Weighted Semi-NMF Algorithms Comparison* provides an overview of the semi-nmf algorithm and part of the update equation for ws-nmf. *Convex and Semi-Nonnegative Matrix Factorizations* provides details on a complete algorithm for the non-weighted semi-nmf algorithm. Using these details we developed a complete update algorithm which may not yet have been published, further literature review is necessary to determine the novelty of the algorithm.

As in the ls-nmf algorithm, the uncertainties are converted to weights Uw. In both algorithms, the loss function remains the same and uses the uncertainty and not the weights to maintain consistency with PMF5. The update equations for ws-nmf are:

$$W_{t+1,i} = (H^T Uw_i^d H)^{-1}(H^T Uw_i^d V_i)$$

$$H_{t+1,i} = H_{t,i}\sqrt{\frac{((V^T Uw)W_{t+1})_i^+ + [H_t(W_{t+1}^T UwW)^-]_i}{((V^T Uw)W_{t+1})_i^- + [H_t(W_{t+1}^T UwW)^+]_i}}$$

Each matrix requires a separate calculation for each sample=N for W and each feature=M for H and is indicated by the i index, increasing the computational complexity of the algorithm. Uw_i^d is the diagonal matrix created from the ith column or row of Uw depending on which matrix is being updated. The first section of the update equation for $W_{t+1,i}$ requires calculating the inverse which is only possible if the determinant is not equal to zero, in which case the pseudo-inverse is used. The calculation of H_{t+1} requires several additional steps. The positive and negative values from W are separated with $W^- = \frac{(|W|-W)}{2.0}$ and $W^+ = \frac{(|W|+W)}{2.0}$.

Optimization

The algorithms are intended to be used directly as a python package, through Jupyter notebooks and eventually through a web application. With this in mind, the performance of the code must be considered during implementation. Here are a few approaches that have been taken to optimize the code/algorithm (with performance metrics shown later):

1. Using Python 3.11, which has performance increases of 10-60% over 3.10
2. Use of parallelization for batch modeling, fitting multiple models at a time.
3. The algorithms have also be written in Rust, a low level language, which provides increased memory efficiency and decreased runtime.

LS-NMF Example

Here is an example of how to use the code and generate either a single or multiple models

In [1]:
```
# Notebook imports
import os
import sys
import copy
import logging
import time
import json
import pandas as pd
import numpy as np
import plotly
import plotly.graph_objects as go
import plotly.figure_factory as ff
import plotly.express as px
from plotly.subplots import make_subplots

module_path = os.path.abspath(os.path.join('..'))
if module_path not in sys.path:
    sys.path.append(module_path)
```

Sample Datasets

PMF5 comes with three sample datasets which we will use in the code demo.

In [2]:
```
# Baton Rouge Dataset
br_input_file = os.path.join("D:\\", "projects", "nmf_py", "data", "Dataset-BatonRouge-con.csv")
br_uncertainty_file = os.path.join("D:\\", "projects", "nmf_py", "data", "Dataset-BatonRouge-unc.csv")
br_output_path = os.path.join("D:\\", "projects", "nmf_py", "output", "BatonRouge")
# Baltimore Dataset
b_input_file = os.path.join("D:\\", "projects", "nmf_py", "data", "Dataset-Baltimore_con.txt")
b_uncertainty_file = os.path.join("D:\\", "projects", "nmf_py", "data", "Dataset-Baltimore_unc.txt")
b_output_path = os.path.join("D:\\", "projects", "nmf_py", "output", "Baltimore")
# Saint Louis Dataset
sl_input_file = os.path.join("D:\\", "projects", "nmf_py", "data", "Dataset-StLouis-con.csv")
sl_uncertainty_file = os.path.join("D:\\", "projects", "nmf_py", "data", "Dataset-StLouis-unc.csv")
sl_output_path = os.path.join("D:\\", "projects", "nmf_py", "output", "StLouis")
```

Code Imports

We import the modules for the model and a datahandler.

In [3]:
```
from src.data.datahandler import DataHandler
from src.model.nmf import NMF
from src.model.batch_nmf import BatchNMF
```

Parameters

In [4]:
```
index_col = "Date"                  # the index of the input/uncertainty datasets
factors = 6                         # the number of factors
method = "ls-nmf"                   # "ls-nmf", "ws-nmf"
init_method = "col_means"           # default is column means "col_means", "kmeans", "cmeans"
init_norm = True                    # if init_method is either kmeans or cmeans, whether to normalize the data prior to clustering.
seed = 42                           # seed = 26586, most comparable model to PMF5 currently found
max_iterations = 20000              # the maximum number of iterations for fitting a model
converge_delta = 0.1                # convergence criteria for the change in loss, Q
converge_n = 10                     # convergence criteria for the number of steps where the loss changes by less than converge_del
dataset = "br"                      # "br": Baton Rouge, "b": Baltimore, "sl": St Louis
verbose = True                      # adds more verbosity to the algorithm workflow on execution.
optimized = True                    # use the Rust code if possible
```

Load the Data

In [5]:
```
# Loading the Baton Rouge dataset
dh_br = DataHandler(
    input_path=br_input_file,
    uncertainty_path=br_uncertainty_file,
    index_col=index_col,
    sn_threshold=2.0
)
V_br, U_br = dh_br.get_data()           # Cleaned input dataset (numpy array), Cleaned uncertainty dataset (numpy array)
```

```
18-Jan-24 13:06:39 - Input and output configured successfully
```

Initialize and Train

In [6]:
```
# Training a single model
nmf_br = NMF(V=V_br, U=U_br, factors=factors, method=method, seed=seed, optimized=optimized, verbose=verbose)
nmf_br.initialize(init_method=init_method, init_norm=init_norm, fuzziness=5.0)
nmf_br.train(max_iter=max_iterations, converge_delta=converge_delta, converge_n=converge_n)
```

```
18-Jan-24 13:06:41 - Model: -1, Seed: 42, Q(true): 68313.7773, Q(robust): 58644.709, Steps: 1956/20000, Converged: True, Runtime: 1.
32 sec
```

Here a single model was created that used the optimized Rust code with the ws-nmf algorithm. 940 iterations were taken before the convergence criteria was met with a resulting loss value of $Q = 84264.11$.

In [7]:
```
%%time
# Training multiple models
models = 10                   # number of models to create
parallel = True               # execute training in parallel

batch_br = BatchNMF(V=V_br, U=U_br, factors=factors, models=models, method=method, seed=seed,
                    init_method=init_method, init_norm=init_norm,
                    max_iter=max_iterations, converge_delta=converge_delta,
                    converge_n=converge_n, parallel=parallel, optimized=optimized,
                    verbose=verbose
                   )
batch_br.train()
```

```
18-Jan-24 13:06:46 - Model: 1, Q(true): 65069.5744, Q(robust): 55644.9141, Seed: 8925, Converged: True, Steps: 2300/20000
18-Jan-24 13:06:46 - Model: 2, Q(true): 65541.8666, Q(robust): 55585.2486, Seed: 77395, Converged: True, Steps: 1291/20000
18-Jan-24 13:06:46 - Model: 3, Q(true): 65049.2052, Q(robust): 55589.223, Seed: 65457, Converged: True, Steps: 1348/20000
18-Jan-24 13:06:46 - Model: 4, Q(true): 66051.8651, Q(robust): 56544.1801, Seed: 43887, Converged: True, Steps: 1134/20000
18-Jan-24 13:06:46 - Model: 5, Q(true): 63928.0347, Q(robust): 54497.1779, Seed: 43301, Converged: True, Steps: 1495/20000
18-Jan-24 13:06:46 - Model: 6, Q(true): 63840.4524, Q(robust): 54423.8403, Seed: 85859, Converged: True, Steps: 2343/20000
18-Jan-24 13:06:46 - Model: 7, Q(true): 66533.3494, Q(robust): 56896.0526, Seed: 8594, Converged: True, Steps: 1011/20000
18-Jan-24 13:06:46 - Model: 8, Q(true): 65055.9772, Q(robust): 55644.0452, Seed: 69736, Converged: True, Steps: 2167/20000
18-Jan-24 13:06:46 - Model: 9, Q(true): 63894.2365, Q(robust): 54459.2191, Seed: 20146, Converged: True, Steps: 2286/20000
18-Jan-24 13:06:46 - Model: 10, Q(true): 66038.0929, Q(robust): 56519.5267, Seed: 9417, Converged: True, Steps: 1677/20000
18-Jan-24 13:06:46 - Results - Best Model: 6, Q(true): 63840.4524, Q(robust): 54423.8403, Converged: True
18-Jan-24 13:06:46 - Runtime: 0.08 min(s)
CPU times: total: 1.39 s
Wall time: 4.95 s
```

Out[7]:
```
(True, '')
```

In [8]:
```
# Save results
# batch_name = f"test-batch-f{factors}"
# br_full_output_path = os.path.join("..", "..", "..", "data", "output")
# batch_br.save(batch_name=batch_name, output_directory=br_full_output_path, pickle_batch=False, header=dh_br.features)
```

In [9]:
```
# Imports for comparing to PMF5 outputs
from tests.factor_comparison import FactorComp
from src.utils import calculate_Q
```

In [10]:
```
# Compare the results to the PMF5 output on the same dataset and the same number of factors
br_pmf_profile_file = os.path.join("D:\\", "projects", "nmf_py", "data", "factor_test", f"br{factors}f_profiles.txt")
br_pmf_contribution_file = os.path.join("D:\\", "projects", "nmf_py", "data", "factor_test", f"br{factors}f_contributions.txt")
br_pmf_residuals_file = os.path.join("D:\\", "projects", "nmf_py", "data", "factor_test",
                                     f"br{factors}f_residuals.txt")
br_profile_comparison = FactorComp(batch_nmf=batch_br, pmf_profile_file=br_pmf_profile_file,
                                   pmf_contribution_file=br_pmf_contribution_file, factors=factors,
                                   features=dh_br.features, residuals_path=br_pmf_residuals_file)
br_pmf_q = calculate_Q(br_profile_comparison.pmf_residuals.values, dh_br.uncertainty_data_processed)
br_profile_comparison.compare(PMF_Q=br_pmf_q)
```

```
Calculating correlation between factors from each epoch: 100%|████████████████████| 10/10 [00:00<00:00, 15.53it/s]
Number of permutations for 6 factors: 720
Calculating average correlation for all permutations for each epoch: 100%|████████████| 10/10 [00:36<00:00,  3.66s/it]
R2 - Model: 4, Best permutations: ['Factor 1', 'Factor 2', 'Factor 4', 'Factor 3', 'Factor 5', 'Factor 6'], Average R2: 0.9824235591
369881,
Profile R2 Avg: 0.9982128176460442, Contribution R2 Avg: 0.9669323425344882, WH R2 Avg: 0.9821255172304316
Profile R2: [0.9999105473090671, 0.9964333821721015, 0.9991743875437269, 0.997882039478663, 0.9965428938349589, 0.9993336555377472],
Contribution R2: [0.9857350637800082, 0.9651437188264308, 0.9688929688927129, 0.9710191489603955, 0.9856446696769423, 0.925158485070
4394],
WH R2: [0.9878041694450078, 0.9781218790703419, 0.9759456998648115, 0.9871898739887672, 0.9873558392754626, 0.9763356417381981]

PMF5 Q(true): 64509.9296875, NMF-PY Model 4 Q(true): 63928.03469764667
```

Here we see that the most comparable NMF-PY model was model 5 from our batch of 10, with an average $R^2 = 0.848$ with a the breakdown for each factor and each of the matrices, plus the product matrix WH. Additionally, we are provided the factor mapping from the NMF-PY model to the PMF5 model, since the order of factors is never guaranteed to match. We are only comparing the outputs of NMF-PY to the single best performing model of PMF5.

In [11]:
```
%%time
# Training multiple models using ls-nmf
models = 10                    # number of models to create
parallel = True                # execute training in parallel
method2 = "ls-nmf"

batch_br2 = BatchNMF(V=V_br, U=U_br, factors=factors, models=models, method=method2, seed=seed,
                     init_method=init_method, init_norm=init_norm,
                     max_iter=max_iterations, converge_delta=converge_delta,
                     converge_n=converge_n, parallel=parallel, optimized=optimized,
                     verbose=verbose
                    )
batch_br2.train()
# br2_full_output_path = f"br2_nmf-output-f{factors}.json"
# batch_br2.save(output_name=br2_full_output_path)
```

```
18-Jan-24 13:07:29 - Model: 1, Q(true): 65069.5744, Q(robust): 55644.9141, Seed: 8925, Converged: True, Steps: 2300/20000
18-Jan-24 13:07:29 - Model: 2, Q(true): 65541.8666, Q(robust): 55585.2486, Seed: 77395, Converged: True, Steps: 1291/20000
18-Jan-24 13:07:29 - Model: 3, Q(true): 65049.2052, Q(robust): 55589.223, Seed: 65457, Converged: True, Steps: 1348/20000
18-Jan-24 13:07:29 - Model: 4, Q(true): 66051.8651, Q(robust): 56544.1801, Seed: 43887, Converged: True, Steps: 1134/20000
18-Jan-24 13:07:29 - Model: 5, Q(true): 63928.0347, Q(robust): 54497.1779, Seed: 43301, Converged: True, Steps: 1495/20000
18-Jan-24 13:07:29 - Model: 6, Q(true): 63840.4524, Q(robust): 54423.8403, Seed: 85859, Converged: True, Steps: 2343/20000
18-Jan-24 13:07:29 - Model: 7, Q(true): 66533.3494, Q(robust): 56896.0526, Seed: 8594, Converged: True, Steps: 1011/20000
18-Jan-24 13:07:29 - Model: 8, Q(true): 65055.9772, Q(robust): 55644.0452, Seed: 69736, Converged: True, Steps: 2167/20000
18-Jan-24 13:07:29 - Model: 9, Q(true): 63894.2365, Q(robust): 54459.2191, Seed: 20146, Converged: True, Steps: 2286/20000
18-Jan-24 13:07:29 - Model: 10, Q(true): 66038.0929, Q(robust): 56519.5267, Seed: 9417, Converged: True, Steps: 1677/20000
18-Jan-24 13:07:29 - Results - Best Model: 6, Q(true): 63840.4524, Q(robust): 54423.8403, Converged: True
18-Jan-24 13:07:29 - Runtime: 0.08 min(s)
CPU times: total: 188 ms
Wall time: 5.03 s
```

Out[11]: (True, '')

In [12]:
```
br2_profile_comparison = FactorComp(batch_nmf=batch_br2, pmf_profile_file=br_pmf_profile_file,
                                    pmf_contribution_file=br_pmf_contribution_file, factors=factors,
                                    features=dh_br.features, residuals_path=br_pmf_residuals_file)
br2_pmf_q = calculate_Q(br2_profile_comparison.pmf_residuals.values, dh_br.uncertainty_data_processed)
br2_profile_comparison.compare(PMF_Q=br2_pmf_q)
```

```
Calculating correlation between factors from each epoch: 100%|████████████████████| 10/10 [00:00<00:00, 15.90it/s]
Number of permutations for 6 factors: 720
Calculating average correlation for all permutations for each epoch: 100%|██████████| 10/10 [00:36<00:00,  3.63s/it]
R2 - Model: 4, Best permutations: ['Factor 1', 'Factor 2', 'Factor 4', 'Factor 3', 'Factor 5', 'Factor 6'], Average R2: 0.9824235591
369881,
Profile R2 Avg: 0.9982128176460442, Contribution R2 Avg: 0.9669323425344882, WH R2 Avg: 0.9821255172304316
Profile R2: [0.9999105473090671, 0.9964333821721015, 0.9991743875437269, 0.997882039478663, 0.9965428938349589, 0.9993336555377472],
Contribution R2: [0.9857350637800082, 0.9651437188264308, 0.9688929688927129, 0.9710191489603955, 0.9856446696769423, 0.925158485070
4394],
WH R2: [0.9878041694450078, 0.9781218790703419, 0.9759456998648115, 0.9871898739887672, 0.9873558392754626, 0.9763356417381981]

PMF5 Q(true): 64509.9296875, NMF-PY Model 4 Q(true): 63928.03469764667
```

Analysis

For a single dataset/factor count model:

Graphic 4 factor timeseries by feature: Line graph showing the timeseries of the concentrations for each factor compared to PMF5.

Graphic 5 output residual: line graphs PMF5 to NMF-PY

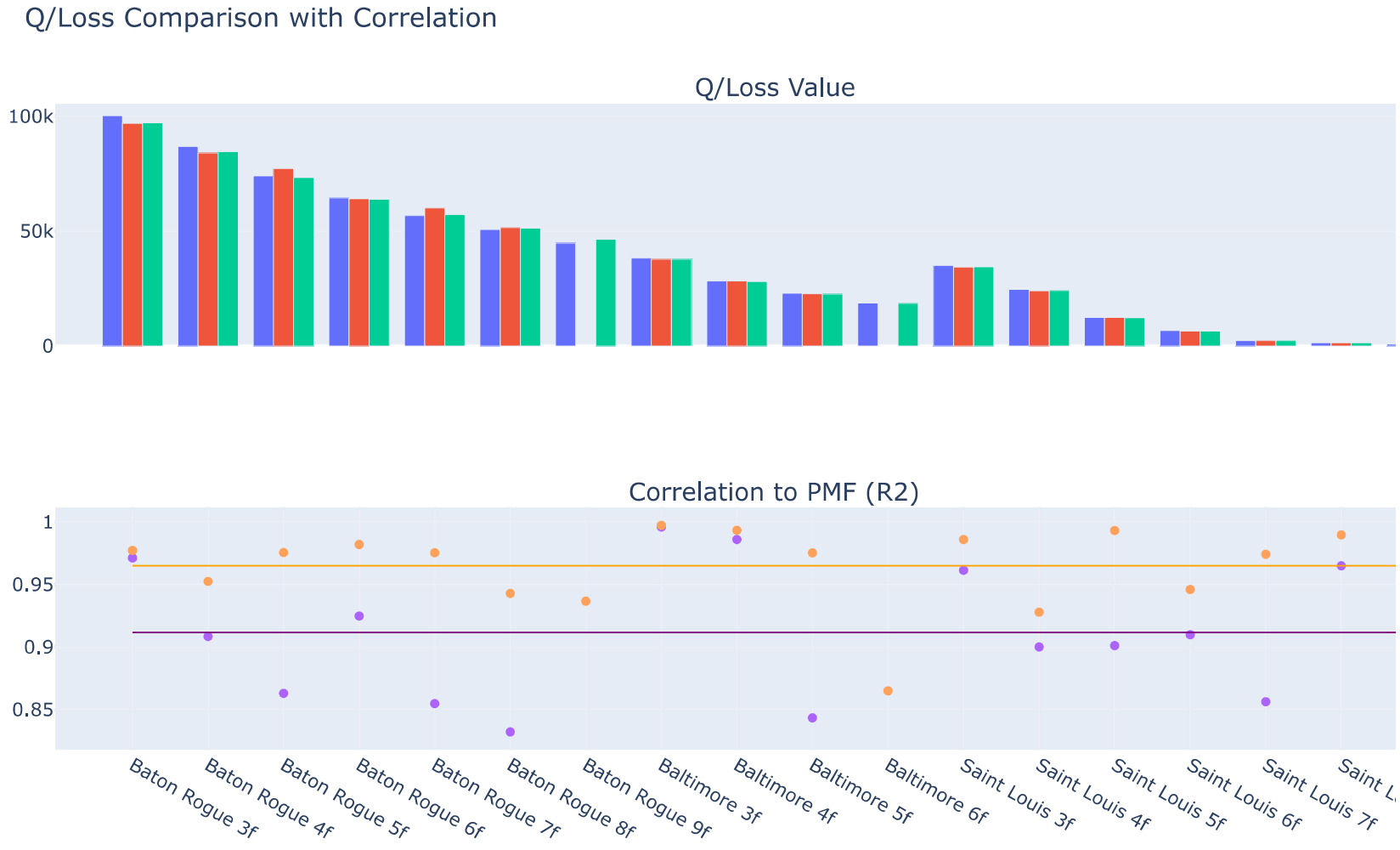
Q/Loss Comparison

Here the three different datasets were tested using both algorithms. 200 models were created for each dataset, each factor count, and each algorithm. The most compariable Q and R^2 are shown, along with the best performing (lowest Q) model Q is shown. The NaN were models that had not yet been completed. Each model requires a comparison of all permutations of the factor ordering to the PMF5 output, making comparison of factors > 5 time consuming to complete for 200 models. The R^2 value is the total average, the comparison of the factor profile (H matrix), the factor contribution (W matrix), and the model concentration output (WH).

In [13]:
```
q_analysis_file = "q_analysis.json"
q_analysis = {}
with open(q_analysis_file, "r") as data_file:
    q_analysis = json.load(data_file)
q_df = pd.DataFrame(q_analysis.values())
q_df["dataset"].replace({"br": "Baton Rogue", "b": "Baltimore", "sl": "Saint Louis"}, inplace=True)
q_df.rename(columns={"Q(ls-nmf-R2)": "R2(ls-nmf)", "Q(ws-nmf-R2)": "R2(ws-nmf)"}, inplace=True)
q_df = q_df.round(decimals=4)
q_fig = ff.create_table(q_df)
q_fig.update_layout(width=1200, height=400)
q_fig.show()
```

dataset	factors	Q(pmf)	Q(ls-nmf)	Q(ls-nmf-min)	R2(ls-nmf)	Q(ws-nmf)	Q
Baton Rogue	3	100209.2188	97137.6125	97111.315	0.9773	96971.7436	96
Baton Rogue	4	86895.6875	84605.5486	83682.7681	0.9525	84062.1961	83
Baton Rogue	5	74038.875	73319.4542	73070.0939	0.9756	77295.756	73
Baton Rogue	6	64509.9297	63834.4389	63823.7798	0.982	64081.0649	64
Baton Rogue	7	56798.7773	57233.6384	57130.5282	0.9754	60068.7058	57
Baton Rogue	8	50683.7969	51274.6023	50943.287	0.9429	51587.4397	51
Baton Rogue	9	44813.9102	46483.5281	45079.4151	0.9367	nan	na
Baltimore	3	38246.6094	37840.6271	37836.5971	0.9973	37830.4964	37
Baltimore	4	28339.4922	28075.9861	28067.5867	0.9934	28360.1729	28
Baltimore	5	22951.4492	22642.0168	22614.34	0.9753	22707.521	22
Baltimore	6	18685.6719	18542.8412	17716.9693	0.865	nan	na
Saint Louis	3	35010.4844	34488.7686	34482.4238	0.986	34394.2224	34
Saint Louis	4	24621.4648	24109.1421	21368.8941	0.9279	24029.3192	21
Saint Louis	5	12388.043	12280.8189	12279.0721	0.9932	12394.8834	12
Saint Louis	6	6750.7256	6529.8377	6529.0748	0.946	6532.566	65
Saint Louis	7	2333.0437	2392.3925	2374.3632	0.9742	2381.1895	23
Saint Louis	8	1341.9539	1388.6649	1385.1241	0.9897	1398.4528	13
Saint Louis	9	769.1725	792.763	786.7842	0.9814	nan	na

In [14]:
```
labels = q_df["dataset"] + " " + q_df["factors"].astype(str) + "f"
q_df["index"] = labels

q_sub = make_subplots(rows=2, cols=1, shared_xaxes=True, subplot_titles=("Q/Loss Value", "Correlation to PMF (R2)"))

# runtime_df = runtime_df.sort_values(by=["dataset", "factors"], ascending=False)
pmf_trace = go.Bar(x=q_df["index"], y=q_df["Q(pmf)"], name="PMF")
ws_trace = go.Bar(x=q_df["index"], y=q_df["Q(ws-nmf)"], name="WS-NMF")
ls_trace = go.Bar(x=q_df["index"], y=q_df["Q(ls-nmf)"], name="LS-NMF")
q_sub.add_trace(pmf_trace, 1, 1)
q_sub.add_trace(ws_trace, 1, 1)
q_sub.add_trace(ls_trace, 1, 1)

ws_trace2 = go.Scatter(x=q_df["index"], y=q_df["R2(ws-nmf)"], name="R2 WS-NMF", mode='markers')
ls_trace2 = go.Scatter(x=q_df["index"], y=q_df["R2(ls-nmf)"], name="R2 LS-NMF", mode='markers')
ws_mean2 = go.Scatter(x=q_df["index"], y=[q_df["R2(ws-nmf)"].mean()] * len(labels), name="R2 WS-NMF Mean", mode='lines', line=dict(
ls_mean2 = go.Scatter(x=q_df["index"], y=[q_df["R2(ls-nmf)"].mean()] * len(labels), name="R2 LS-NMF Mean", mode='lines', line=dict(

q_sub.add_trace(ws_trace2, row=2, col=1)
q_sub.add_trace(ls_trace2, row=2, col=1)
q_sub.add_trace(ws_mean2, row=2, col=1)
q_sub.add_trace(ls_mean2, row=2, col=1)

q_sub.update_layout(height=600, width=1200, title_text="Q/Loss Comparison with Correlation")
q_sub.show()

print(f"Mean R2 - ls-nmf: {round(q_df['R2(ls-nmf)'].mean(),4)}, ws-nmf: {round(q_df['R2(ws-nmf)'].mean(),4)}")
```




```
Mean R2 - ls-nmf: 0.9651, ws-nmf: 0.9116
```

In [15]:
```
q_dif = copy.copy(q_df)
q_dif["Q dif(ls-nmf)"] = q_dif["Q(pmf)"] / q_df["Q(ls-nmf)"]
q_dif["Q dif(ws-nmf)"] = q_dif["Q(pmf)"] / q_df["Q(ws-nmf)"]
q_dif = q_dif.round(decimals=4)

q_dif_fig = ff.create_table(q_dif[["dataset", "factors", "R2(ls-nmf)", "R2(ws-nmf)", "Q dif(ls-nmf)", "Q dif(ws-nmf)"]])
q_dif_fig.update_layout(width=1200, height=400)
```

```
q_dif_fig.show()
print(f"Min Dif - ls-nmf: {q_dif['Q dif(ls-nmf)'].min()}, ws-nmf: {q_dif['Q dif(ws-nmf)'].min()}")
print(f"Max Dif - ls-nmf: {q_dif['Q dif(ls-nmf)'].max()}, ws-nmf: {q_dif['Q dif(ws-nmf)'].max()}")
```

dataset	factors	R2(ls-nmf)	R2(ws-nmf)	Q dif(ls-nmf)
Baton Rogue	3	0.9773	0.9713	1.0316
Baton Rogue	4	0.9525	0.9085	1.0271
Baton Rogue	5	0.9756	0.863	1.0098
Baton Rogue	6	0.982	0.9248	1.0106
Baton Rogue	7	0.9754	0.8548	0.9924
Baton Rogue	8	0.9429	0.8322	0.9885
Baton Rogue	9	0.9367	nan	0.9641
Baltimore	3	0.9973	0.996	1.0107
Baltimore	4	0.9934	0.986	1.0094
Baltimore	5	0.9753	0.8434	1.0137
Baltimore	6	0.865	nan	1.0077
Saint Louis	3	0.986	0.9614	1.0151
Saint Louis	4	0.9279	0.9001	1.0213
Saint Louis	5	0.9932	0.9012	1.0087
Saint Louis	6	0.946	0.9099	1.0338
Saint Louis	7	0.9742	0.8563	0.9752
Saint Louis	8	0.9897	0.965	0.9664
Saint Louis	9	0.9814	nan	0.9702

```
Min Dif - ls-nmf: 0.9641, ws-nmf: 0.9456
Max Dif - ls-nmf: 1.0338, ws-nmf: 1.0337
```

Runtime Comparison

Here the runtime performance was measured for creating 10 models having comparable loss. The python code was run on the parallelized Rust functions and all values are in seconds.

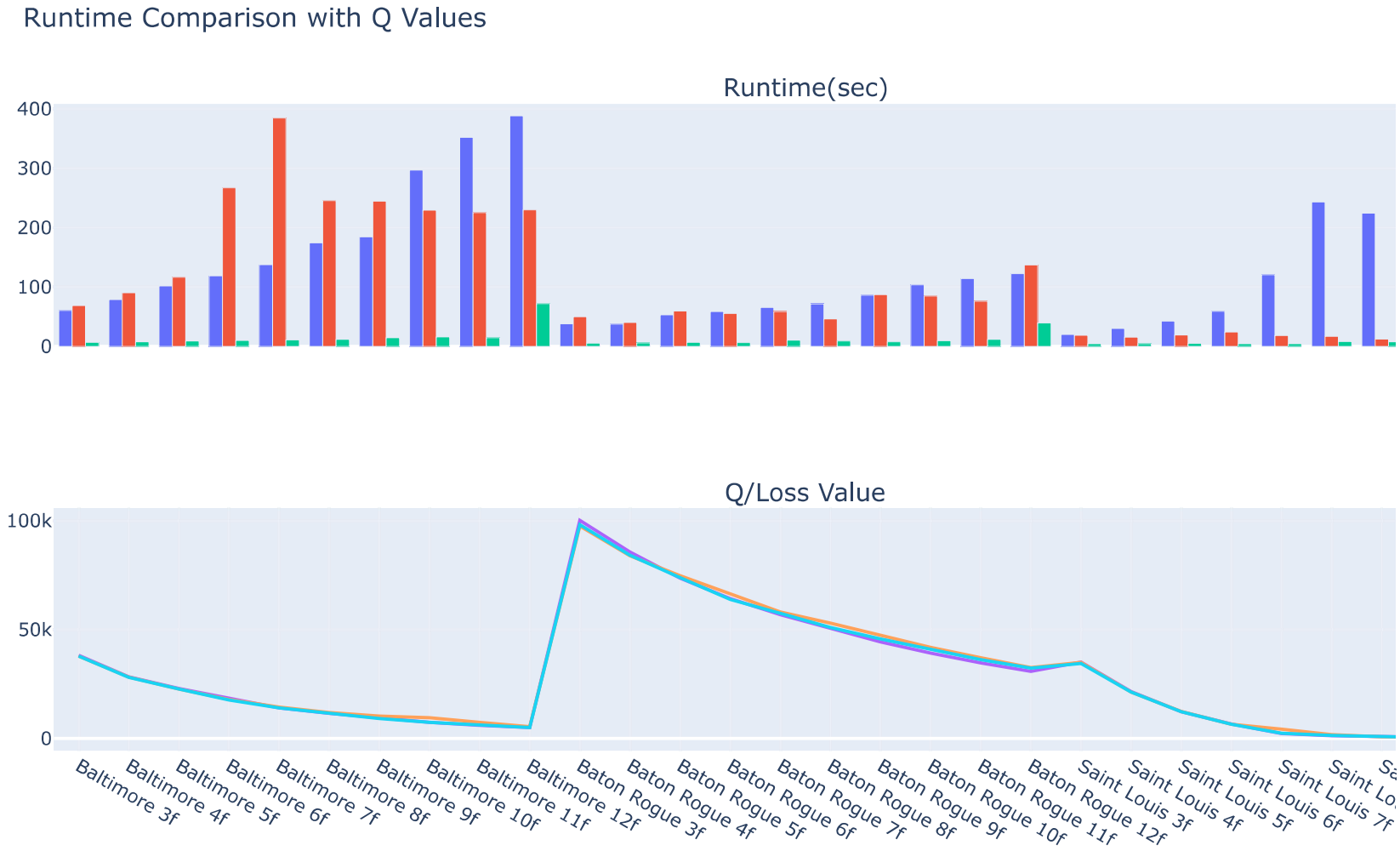
In [16]:
```
runtime_analysis_file = "runtime_analysis.json"
runtime_analysis = {}
with open(runtime_analysis_file, "r") as data_file:
    runtime_analysis = json.load(data_file)
runtime_df = pd.DataFrame(runtime_analysis.values())
runtime_df["dataset"].replace({"br": "Baton Rogue", "b": "Baltimore", "sl": "Saint Louis"}, inplace=True)
runtime_df = runtime_df.sort_values(by=["dataset", "factors"])
runtime_df = runtime_df.round(decimals=2)
runtime_fig = ff.create_table(runtime_df)
```

```
runtime_fig.update_layout(width=1200, height=600)

runtime_fig.show()
```

dataset	factors	ws-nmf-runtime	ls-nmf-runtime	pmf-runtime	ws-nmf-Q	ls-nmf
Baltimore	3	68.76	6.85	60.45	37845.33	37841.
Baltimore	4	90.16	7.73	78.78	28140.49	28081.
Baltimore	5	116.47	9.16	101.69	22709.74	22634.
Baltimore	6	267.08	9.96	118.9	17916.86	17731.
Baltimore	7	384.73	10.86	137.47	14432.26	14010.
Baltimore	8	245.69	11.7	174.21	11946.31	11472.
Baltimore	9	244.52	14.43	184.27	10368.65	9237.9
Baltimore	10	229.56	15.86	296.96	9502.87	7417.2
Baltimore	11	225.53	14.8	352.28	7433.58	6112.7
Baltimore	12	230.14	71.98	388.12	5437.54	5109.4
Baton Rogue	3	49.74	5.47	37.91	97546.0	98090.
Baton Rogue	4	40.32	6.01	37.75	83839.61	84094.
Baton Rogue	5	59.53	6.82	53.1	74760.7	73810.
Baton Rogue	6	55.35	6.45	58.7	66355.73	63840.
Baton Rogue	7	58.85	10.56	65.77	58040.18	57431.
Baton Rogue	8	46.11	9.23	71.73	52989.31	51007.
Baton Rogue	9	86.94	7.82	86.17	47411.22	45699.
Baton Rogue	10	84.76	9.4	103.9	41782.32	40704.
Baton Rogue	11	76.28	11.68	114.4	37120.72	36179.
Baton Rogue	12	137.18	39.18	122.69	32534.71	32374.
Saint Louis	3	18.78	4.33	20.3	35013.7	34485.
Saint Louis	4	15.46	4.62	30.33	21275.76	21373.
Saint Louis	5	19.04	4.91	42.67	12469.15	12284.
Saint Louis	6	24.19	4.46	59.12	6537.61	6533.7
Saint Louis	7	18.05	4.42	120.77	4269.61	2392.1
Saint Louis	8	16.89	8.01	243.08	1771.86	1392.6
Saint Louis	9	12.08	7.68	224.21	804.15	792.32
Saint Louis	10	13.93	7.3	229.5	482.86	434.67
Saint Louis	11	8.49	9.26	280.67	250.68	184.8
Saint Louis	12	12.4	85.1	253.11	0.39	0.36

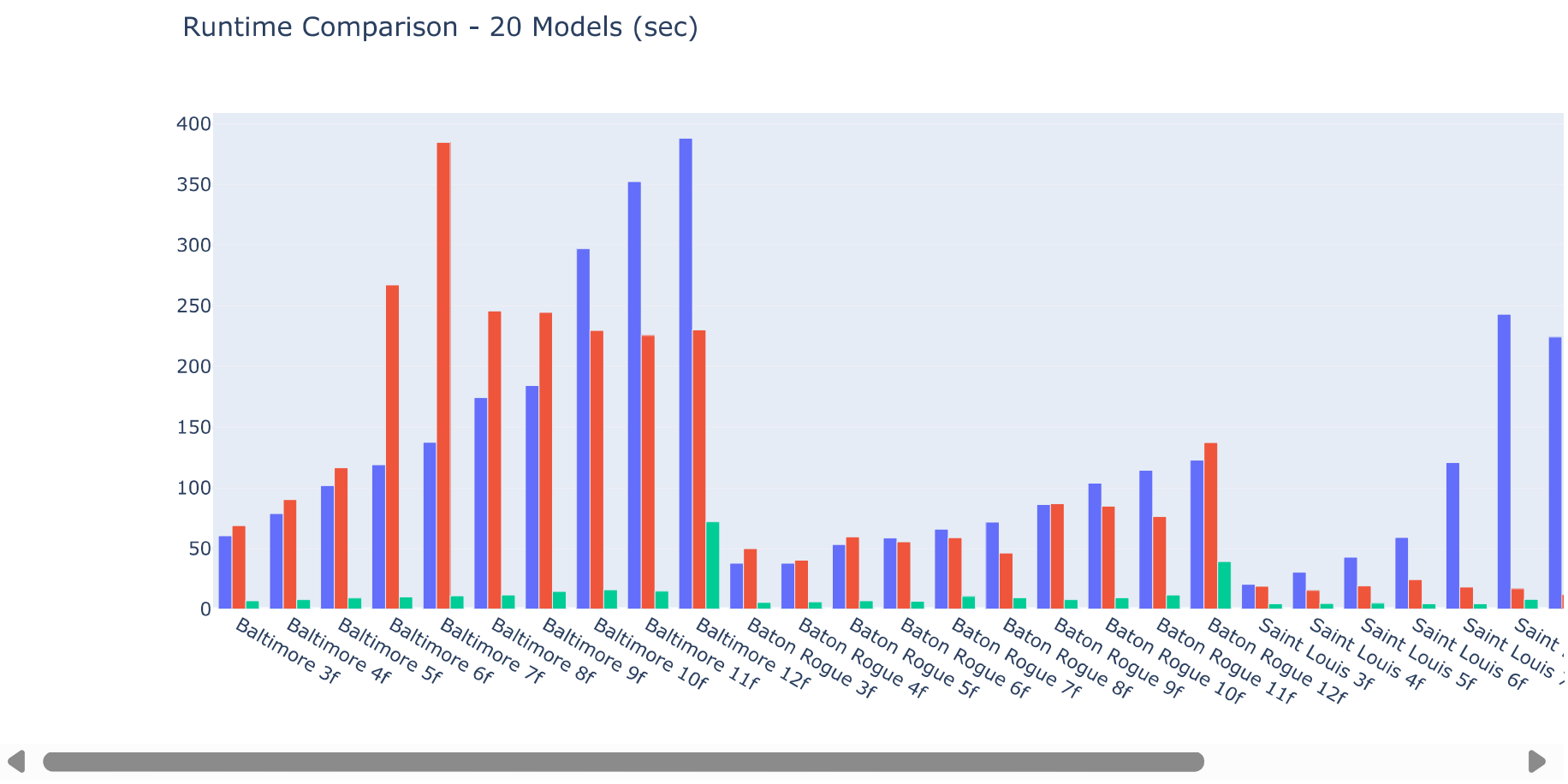
In [17]:
```
labels = runtime_df["dataset"] + " " + runtime_df["factors"].astype(str) + "f"
runtime_df["index"] = labels

runtime_sub = make_subplots(rows=2, cols=1, shared_xaxes=True, subplot_titles=("Runtime(sec)", "Q/Loss Value"))
```

```
rt_pmf_trace = go.Bar(x=runtime_df["index"], y=runtime_df["pmf-runtime"], name="PMF")
rt_ws_trace = go.Bar(x=runtime_df["index"], y=runtime_df["ws-nmf-runtime"], name="WS-NMF")
rt_ls_trace = go.Bar(x=runtime_df["index"], y=runtime_df["ls-nmf-runtime"], name="LS-NMF")
runtime_sub.add_trace(rt_pmf_trace, 1, 1)
runtime_sub.add_trace(rt_ws_trace, 1, 1)
runtime_sub.add_trace(rt_ls_trace, 1, 1)

pmf_trace2 = go.Scatter(x=runtime_df["index"], y=runtime_df["pmf-Q"], name="Q PMF", mode='lines')
ws_trace2 = go.Scatter(x=runtime_df["index"], y=runtime_df["ws-nmf-Q"], name="Q WS-NMF", mode='lines')
ls_trace2 = go.Scatter(x=runtime_df["index"], y=runtime_df["ls-nmf-Q"], name="Q LS-NMF", mode='lines')
runtime_sub.add_trace(pmf_trace2, row=2, col=1)
runtime_sub.add_trace(ws_trace2, row=2, col=1)
runtime_sub.add_trace(ls_trace2, row=2, col=1)

runtime_sub.update_layout(height=600, width=1200, title_text="Runtime Comparison with Q Values")
runtime_sub.show()
```



In [18]:
```
labels = runtime_df["dataset"] + " " + runtime_df["factors"].astype(str) + "f"
runtime_df["index"] = labels

# runtime_df = runtime_df.sort_values(by=["dataset", "factors"], ascending=False)
pmf_trace = go.Bar(x=runtime_df["index"], y=runtime_df["pmf-runtime"], name="PMF")
ws_trace = go.Bar(x=runtime_df["index"], y=runtime_df["ws-nmf-runtime"], name="WS-NMF")
ls_trace = go.Bar(x=runtime_df["index"], y=runtime_df["ls-nmf-runtime"], name="LS-NMF")
```

```
runtime_fig2 = go.Figure(data=[pmf_trace, ws_trace, ls_trace])
runtime_fig2.update_layout(barmode='group')
runtime_fig2.layout.title = "Runtime Comparison - 20 Models (sec)"
runtime_fig2.layout.height = 500
runtime_fig2.layout.width = 1200
runtime_fig2.show()
```



In [19]:
```
ls_runtime_metric = (runtime_df['ls-nmf-runtime'] / runtime_df['factors']).mean()
ws_runtime_metric = (runtime_df['ws-nmf-runtime'] / runtime_df['factors']).mean()
pmf_runtime_metric = (runtime_df['pmf-runtime'] / runtime_df['factors']).mean()
print(f"Ratio of runtime to factor count - LS-NMF: {round(ls_runtime_metric,3)}, WS-NMF: {round(ws_runtime_metric,3)}, PMF: {round(
```

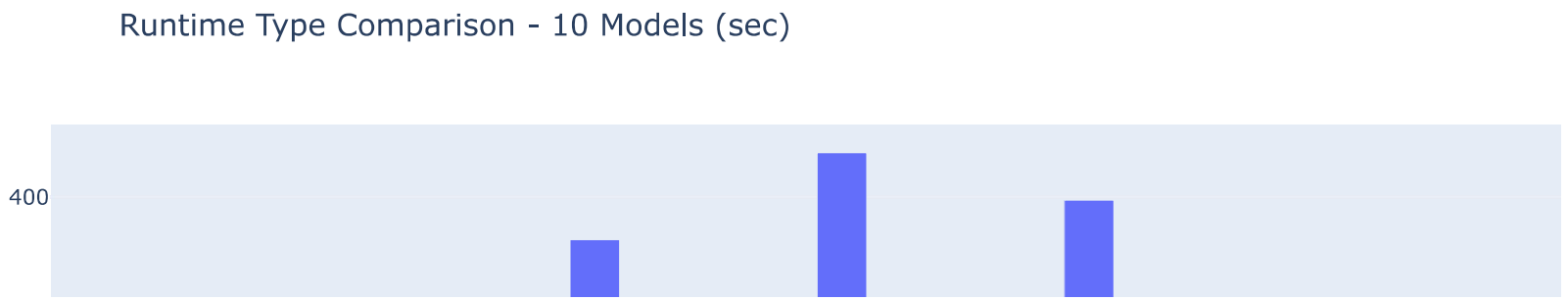
```
Ratio of runtime to factor count - LS-NMF: 1.71, WS-NMF: 13.813, PMF: 17.074
```

In [20]:
```
type_runtime_analysis_file = "type_runtime_analysis.json"
type_runtime_analysis = {}
with open(type_runtime_analysis_file, "r") as data_file:
    type_runtime_analysis = json.load(data_file)
type_runtime_df = pd.DataFrame(type_runtime_analysis.values())
type_runtime_df["dataset"].replace({"br": "Baton Rogue", "b": "Baltimore", "sl": "Saint Louis"}, inplace=True)
type_runtime_df = type_runtime_df.sort_values(by=["dataset", "factors"])
type_runtime_fig = ff.create_table(type_runtime_df)
type_runtime_fig.show()
```

dataset	factors	ls-nmf-rust-runtin
Baltimore	4	22.31
Baltimore	5	22.73
Baltimore	6	54.04
Baltimore	7	33.04
Baltimore	8	10.89

In [21]:
```
type_labels = type_runtime_df["dataset"] + " " + type_runtime_df["factors"].astype(str) + "f"
type_runtime_df["index"] = type_labels

ws_py_trace = go.Bar(x=type_runtime_df["index"], y=type_runtime_df["ws-nmf-py-runtime"], name="WS-NMF(PY)")
```

```
ls_pu_trace = go.Bar(x=type_runtime_df["index"], y=type_runtime_df["ls-nmf-py-runtime"], name="LS-NMF(PY)")
ws_trace = go.Bar(x=type_runtime_df["index"], y=type_runtime_df["ws-nmf-rust-runtime"], name="WS-NMF(RUST)")
ls_trace = go.Bar(x=type_runtime_df["index"], y=type_runtime_df["ls-nmf-rust-runtime"], name="LS-NMF(RUST)")

type_runtime_fig2 = go.Figure(data=[ws_py_trace, ws_trace, ls_pu_trace, ls_trace])
type_runtime_fig2.update_layout(barmode='group')
type_runtime_fig2.layout.title = "Runtime Type Comparison - 10 Models (sec)"
type_runtime_fig2.layout.height = 500
type_runtime_fig2.show()
```



Factor Comparison

Here we look at the best model produced by PMF5 for the Baton Rouge dataset using 6 factors, that is the model with the lowest Q(robust), and compare that model to the most comparable models from the LS-NMF and WS-NMF algorithms. Each algorithm generated 200 models and each of those model's output was compared to the PMF5 model, checking the R^2 of each model factor permutation (since mapping the factor order may

be different from NMF-PY to PMF5). R^2 is calculated for the factor profile (H), the factor contributions (W), and the model product output (WH). The best model is determined as the model with the highest R^2 average.

In [23]:
```
# PMF5 input files are converted into pandas dataframes
factors = 6

pmf_profile_file = os.path.join("D:\\", "projects", "nmf_py", "data", "factor_test", f"br{factors}f_profiles.txt")
pmf_contribution_file = os.path.join("D:\\", "projects", "nmf_py", "data", "factor_test", f"br{factors}f_contributions.txt")
pmf_residuals_file = os.path.join("D:\\", "projects", "nmf_py", "data", "factor_test", f"br{factors}f_residuals.txt")

pmf_output = FactorComp(pmf_profile_file=pmf_profile_file, pmf_contribution_file=pmf_contribution_file, residuals_path=pmf_residual
```

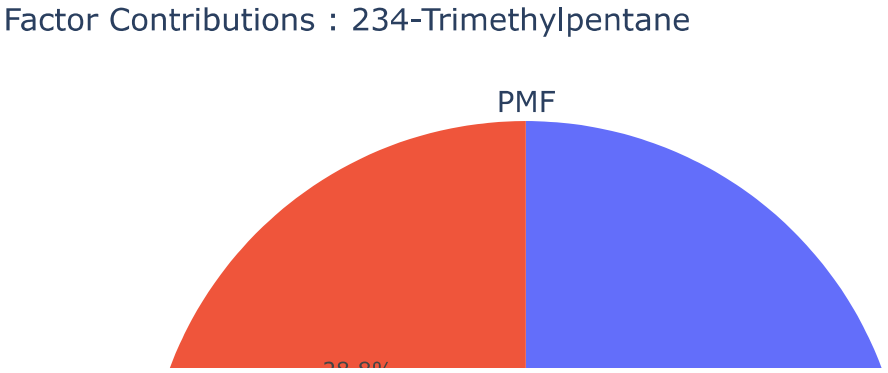
In [24]:
```
# LS-NMF and WS-NMF data pre-processing

profile_comparison_file = os.path.join("profile_compare_analysis_2.json")
profile_json = {}
with open(profile_comparison_file, "r") as p_file:
    profile_json = json.load(p_file)

ls_key = "br-6-ls-nmf"
ws_key = "br-6-ws-nmf"

zero_threshold = 1e-6

species = pmf_output.pmf_profiles_df["species"]
profile_columns = [ f"Factor {i}" for i in range(1, 7)]

ls_mapping = profile_json[ls_key]["factor_mapping"]
ws_mapping = profile_json[ws_key]["factor_mapping"]

ls_profiles = pd.DataFrame(np.array(profile_json[ls_key]["model_profiles"]).T, columns=profile_columns)
ls_profiles[ls_profiles < zero_threshold] = 0.0
ls_profiles["species"] = species
ws_profiles = pd.DataFrame(np.array(profile_json[ws_key]["model_profiles"]).T, columns=profile_columns)
ws_profiles[ws_profiles < zero_threshold] = 0.0
ws_profiles["species"] = species

datetimestamps = pmf_output.pmf_contribution_df["Datetime"].tolist()
ls_contributions = pd.DataFrame(np.array(profile_json[ls_key]["model_contributions"]), columns=profile_columns)
ls_contributions["Datetime"] = datetimestamps
ws_contributions = pd.DataFrame(np.array(profile_json[ws_key]["model_contributions"]), columns=profile_columns)
ws_contributions["Datetime"] = datetimestamps

input_df = dh_br.input_data
```

```
input_df["Date"] = pd.to_datetime(datetimestamps, format="%m/%d/%y %H:%M")
input_df.set_index("Date", inplace=True)
```

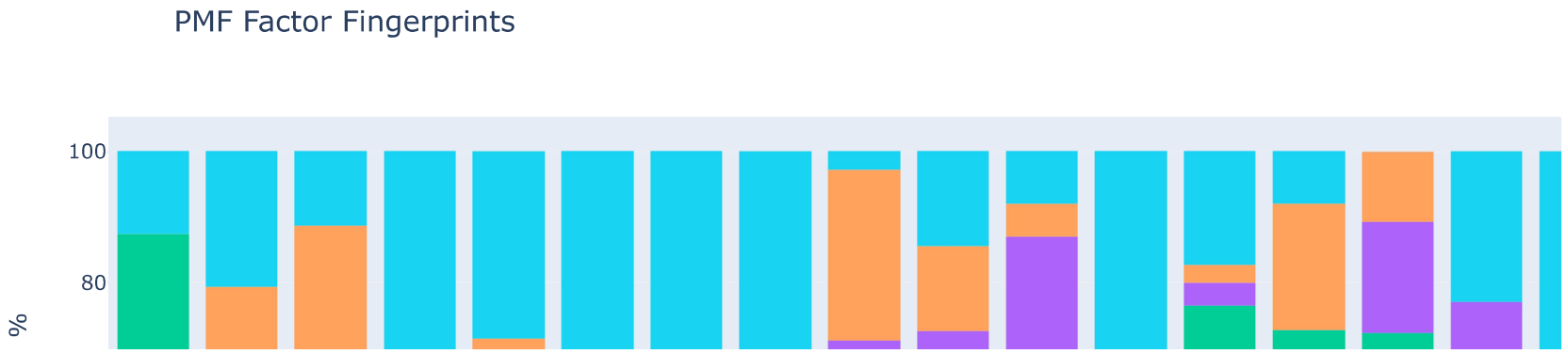
In [30]:
```
from src.data.test_tools import CompareAnalyzer
```

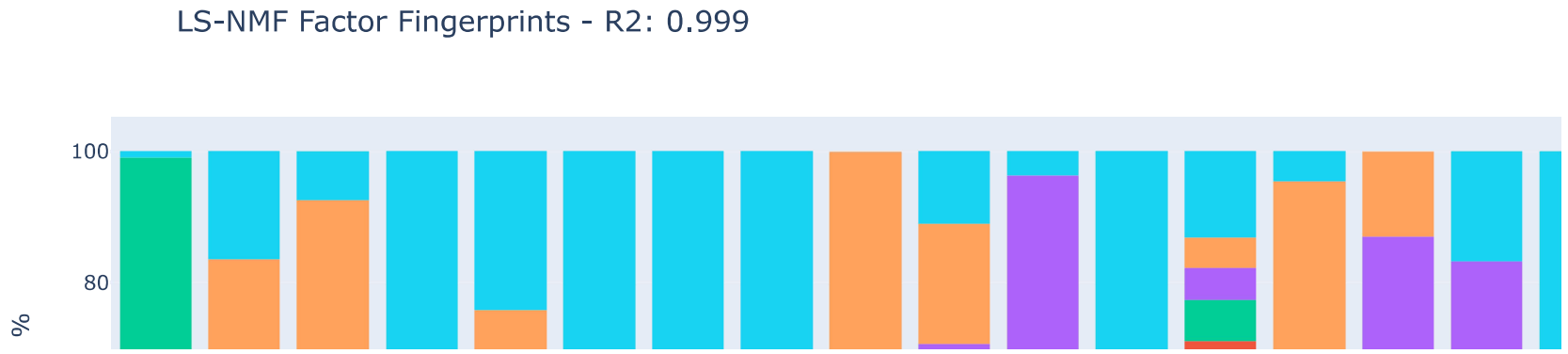
In [32]:
```
ca = CompareAnalyzer(input_df=dh_br.input_data,
                     pmf_profile_df=pmf_output.pmf_profiles_df, ls_profile_df=ls_profiles, ws_profile_df=ws_profiles,
                     pmf_contributions_df=pmf_output.pmf_contribution_df, ls_contributions_df=ls_contributions, ws_contributions_df
                     ls_mapping=ls_mapping, ws_mapping=ws_mapping,
                     features=species,
                     datetimestamps=datetimestamps
                    )
```

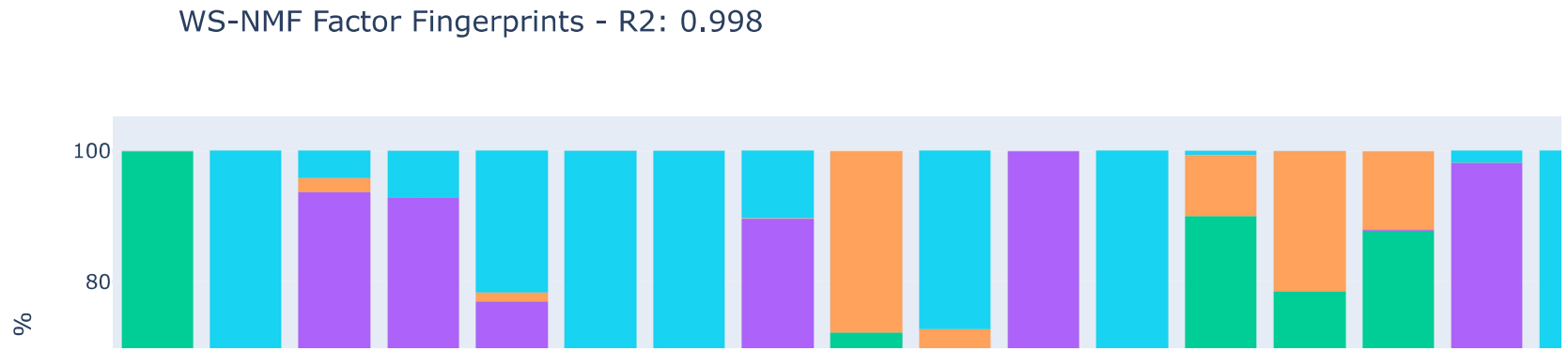
In [33]:
```
ca.plot_factor_contribution(feature_i=2)
```



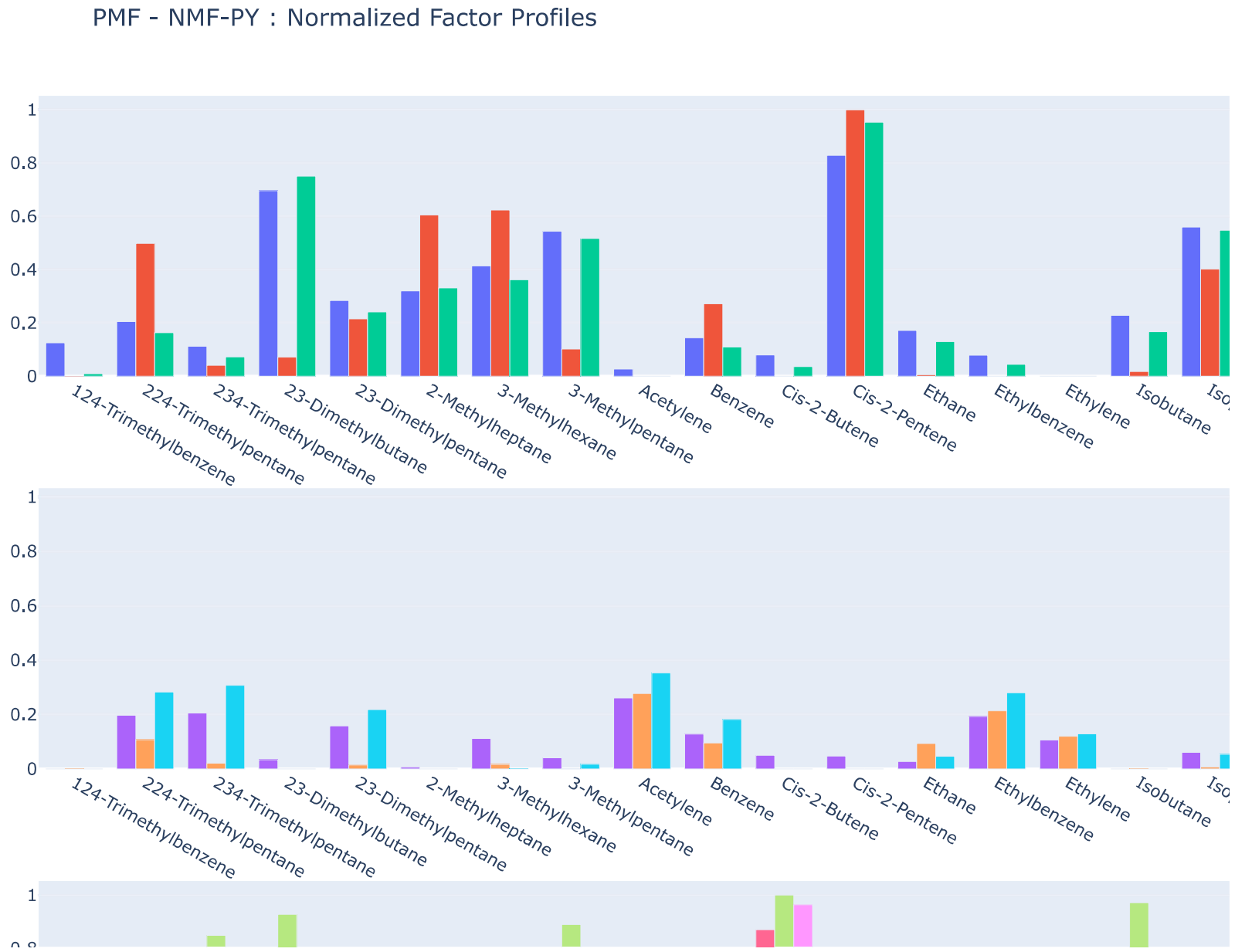
In [34]:
```
ca.plot_fingerprints(ls_nmf_r2=profile_json[ls_key]["model_profile_r2"], ws_nmf_r2=profile_json[ws_key]["model_profile_r2"])
```


PMF Factor Fingerprints
100
80
%

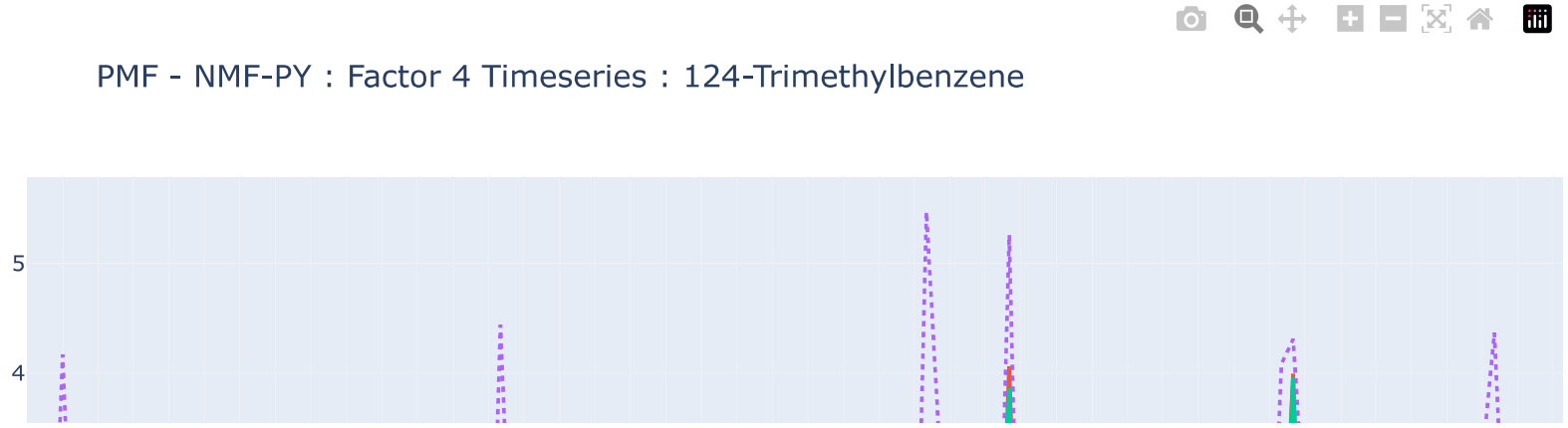

LS-NMF Factor Fingerprints - R2: 0.999
100
80
%

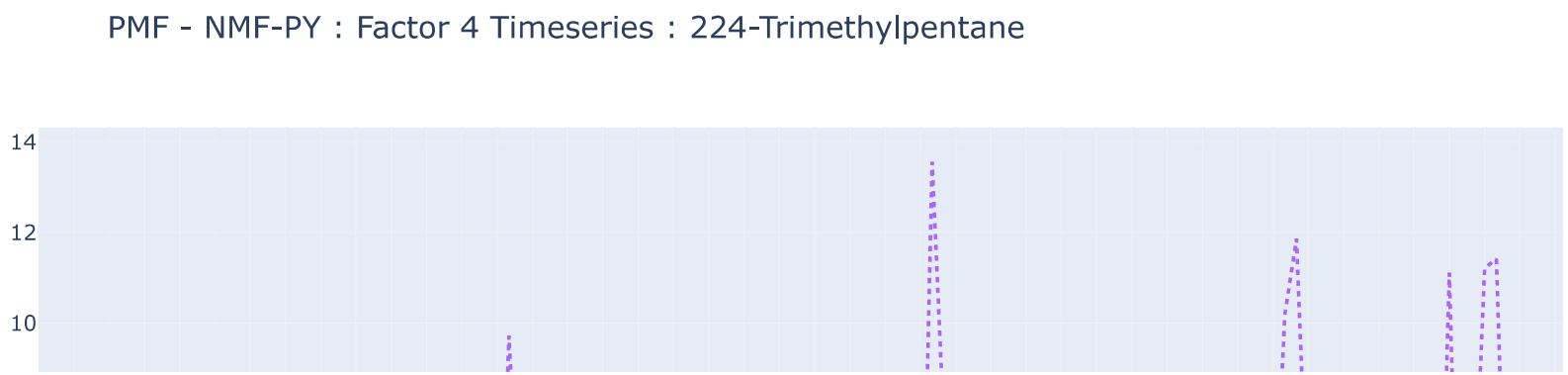


In [35]:
```
ca.plot_factors()
```

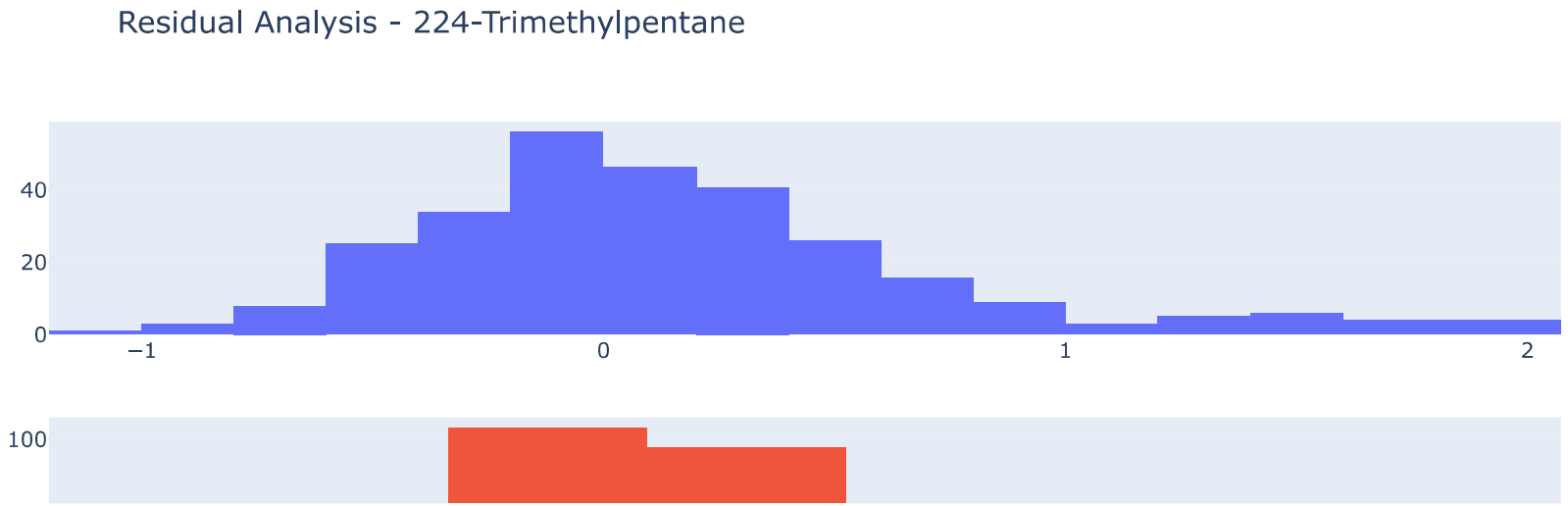

PMF - NMF-PY : Normalized Factor Profiles
1
0.8
0.6
0.4
0.2
0
124-Trimethylbenzene
224-Trimethylpentane
234-Trimethylpentane
23-Dimethylbutane
23-Dimethylpentane
2-Methylheptane
3-Methylhexane
3-Methylpentane
Acetylene
Benzene
Cis-2-Butene
Cis-2-Pentene
Ethane
Ethylbenzene
Ethylene
Isobutane
Iso

In [36]:
```
ca.plot_feature_timeseries(factor_n=3, feature_n=range(0, 2))
```

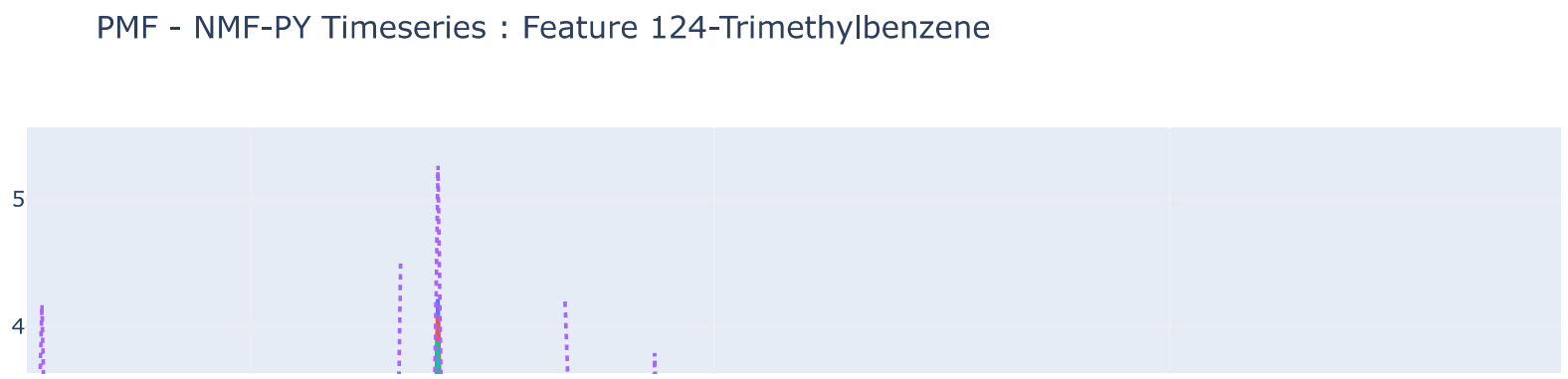

PMF - NMF-PY : Factor 4 Timeseries : 124-Trimethylbenzene
5
4



In [37]:
```
ca.feature_histogram(feature_i=1)
```


Residual Analysis - 224-Trimethylpentane
40
20
0
−1
0
1
2
100

In [38]:
```
ca.timeseries_plot(feature_i=0)
```



In [ ]:

In [ ]: