Monitoring Machine Learning in Python
by Hakim Elakhrass, Maciej Balawejder, and datacamp

Course focus on NannyML
NannyML offers an estimate of the model's performance even when ground truth data is absent
NannyML is an open source Python library that handles detecting data drifts and smartly connects alerts to changes in model performance

Key features
- performance estimation
- find what is broken using univariate and multivariated drift detection along with monitoring data quality
- helps you fix it (example by setting performance based retraining triggers)

First step
NannyML needs the reference (test) dataset and the analysis (production) dataset

Example using a census dataset built into the NannyML library
```
# Import nannyml
import nannyml

# Load US Census Employment dataset
reference, analysis, analysis_gt = nannyml.load_us_census_ma_employment_data()

# Print head of the reference data
print(reference.head())

# Print head of the analysis data
print(analysis.head())
```

Data preparation for NannyML
```
#create data partition for the green taxi dataset
#building an ML model that predicts the tip amount a passenger will leave
#prepared data by only using the fares attached to the credit card since this was the only way to know for certain on tip amounts
#also eliminated the data points in this column that were negative since they likely represented some type of error
data['partition'] = pd.cut(
	data['lpep_pickup_datetime'],
	bins = [pd.to_datetime('2016-12-01'),
```

pd.to_datetime('2016-12-08'),
pd.to_datetime('2016-12-16'),
pd.to_datetime('2017-01-01')],
right=False,
labels= ['train', 'test', 'prod'])

What we do next is split the data into training set, reference set, and analysis set
For our example we use week 1 to train, week 2 to test, and week 3-4 for production

```
# Target column name
target = 'tip_amount'
# Features column name
features = ["PULocationID", "DOLocationID", "trip_distance", "VendorID", "pickup_time"]
```

```
# Train set
X_train = data.loc[data['partition'] == 'train', features]
y_train = data.loc[data['partition'] == 'train', target]

# Test set (later reference set)
X_test = data.loc[data['partition'] == 'test', features]
y_test = data.loc[data['partition'] == 'test', target]

# Production set (later analysis set)
X_prod = data.loc[data['partition'] == 'prod', features]
y_prod = data.loc[data['partition'] == 'prod', target]
```

for our example we will use the lightgbm library to train our dataset
LightGBM (light gradient boosting machine) is known for its efficiency in handling large datasets
which is a go-to solution for predictive modeling tasks

```
# Training the model
model = LGBMRegressor(random_state=42)
model.fit(X_train, y_train)

# Making predictions
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# Evaluating the model on train and test set
mae_train = MAE(y_train, y_pred_train)
mae_test = MAE(y_test, y_pred_test)

# Deploying the model to production
y_pred_prod = model.predict(X_prod)
```

Creating reference and analysis sets
reference period > uses a test set, requires ground truth, this set serves as a baseline for every metric we wish to monitor
analysis period > the latest production data (after the reference period ends), ground truth is optional (NannyML can estimate)

```python
# Creating reference set
reference = X_test.copy() # Test set features
reference['y_pred'] = y_pred_test # Predictions
reference['tip_amount'] = y_test # Labels
reference = reference.join(
    data['lpep_pickup_datetime']) # Timestamp
```

```python
# Creating analysis set
analysis = X_prod.copy() # Production features
analysis['y_pred'] = y_pred_prod # Predictions
analysis = analysis.join(
    data['lpep_pickup_datetime']) # Timestamp
```

Breaking down our reference set

Features						Predictions	Targets	Timestamp
PULocationID	DOLocationID	trip_distance	VendorID	fare_amount	pickup_time	y_pred	tip_amount	lpep_pickup_datetime
112	40	6.93	2	24.5	0	4.920921	5.16	2016-12-08 00:00:00
7	226	1.50	2	12.0	0	2.048210	0.00	2016-12-08 00:00:00
223	223	1.43	2	6.5	0	1.575490	1.56	2016-12-08 00:00:01
112	37	3.25	2	14.5	0	2.810236	2.00	2016-12-08 00:00:03
112	69	10.40	1	36.0	0	7.068890	2.00	2016-12-08 00:00:05

Model ouputs
-predictions > prediction score outputted by the model
-prediction class labels > thresholded probability scores
*for a classification task, there will be an extra column containing prediction class labels (ie thresholded probability scores)

Example:
# Load the dataset
dataset_name = "green_taxi_dataset.csv"
data = pd.read_csv(dataset_name)
features = ['lpep_pickup_datetime', 'PULocationID', 'DOLocationID', 'trip_distance', 'fare_amount', 'pickup_time']

```
target = 'tip_amount'

# Split the training data
X_train = data.loc[data['partition'] == 'train', features]
y_train = data.loc[data['partition'] == 'train', target]

# Split the test data
X_test = data.loc[data['partition'] == 'test', features]
y_test = data.loc[data['partition'] == 'test', target]

# Split the prod data
X_prod = data.loc[data['partition'] == 'prod', features]
y_prod = data.loc[data['partition'] == 'prod', target]

# Fit the model
model = LGBMRegressor(random_state=111, n_estimators=50, n_jobs=1)
model.fit(X_train, y_train)

# Make predictions
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# Deploy the model
y_pred_prod = model.predict(X_prod)

# Create reference and analysis set
reference = X_test.copy() # Test set features
reference['y_pred'] = y_pred_test # Predictions
reference['tip_amount'] = y_test # Labels(ground truth)
reference = reference.join(data['lpep_pickup_datetime']) # Timestamp

analysis = X_prod.copy() # Production features
analysis['y_pred'] = y_pred_prod # Predictions
analysis = analysis.join(data['lpep_pickup_datetime']) # Timestamp
```

Performance estimation
direct loss estimation (DLE) trains an extra ML model to estimate the value of the loss function
*this is the difference between the model's predictions and the actual target values
NannyML uses the LGBM algorithm as the 'extra' ML model

using Python

estimator = nannyml.DLE(
y_true = ‘target’, #this is the ground truth
y_pred = ‘y_pred’, #this is your model’s predictions
metrics = [‘rmse’] #one of six available from nannyml
timestamp_column_name = ‘timestamp’, #need to specifiy the column containing the timestamps
chunk_period = ‘d’, #d stands for daily performance evaluation
feature_column_names=features #features represents a list of column names representing the features used by the model
tune_hyperparameters=False) #default is false, can tune the external model if willing to use the computational power

NannyML’s algorithms operate similarly to scikit
“fit” it using the reference set and then estimate our metrics on the analysis set
results are then stored in a NannyML results object
this can be converted to a pandas dataframe

Using Python
estimator.fit(reference)
results = estimator.estimate(analysis)

Using confidence based performance estimation (CBPE)
used for both binary and multiclass classification problems
works by using the confidence scores of the model’s predictions
with these scores, CBPE estimates all the elements of the confusion matrix
we can then estimate various classification performance metrics such as accuracy, ROC AUC, F1 score, or precision

Using Python
estimator = nannyml.CBPE(
y_pred_proba = ‘y_pred_proba’, #this holds the predicted probabilities
y_pred = ‘y_pred’, #this holds the model’s predicted classes
y_true = ‘targets’, #ground truth
timestamp_column_name = ‘timestamp’.,
metrics = [‘roc_auc’],
chunk_period = ‘d’,
problem_type = ‘classification_binary’) #this indicates whether we have a binary or multiclass classification problem

estimator.fit(reference)
results = estimator.estimate(analysis)

Visualizing results

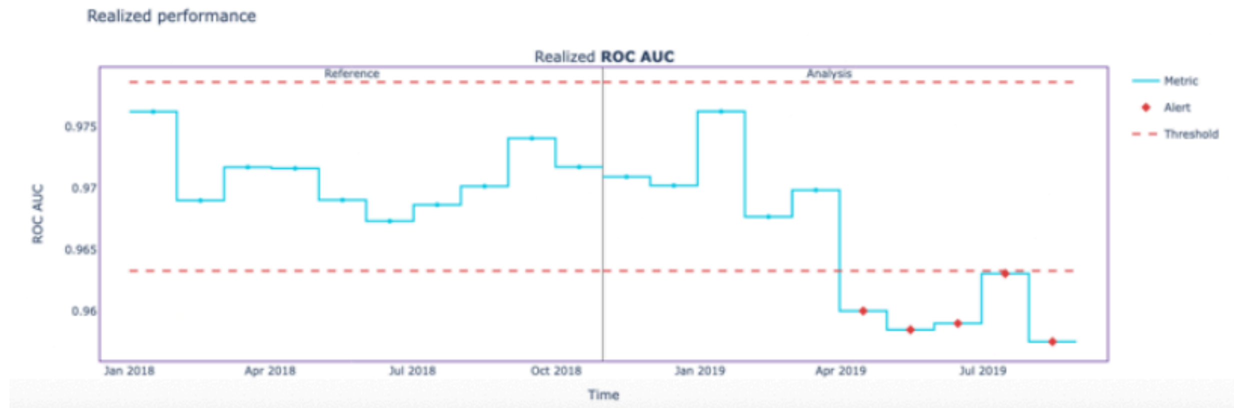
results.plot().show()
example output>

Estimated vs realized performance
estimated > measures how well model is expected to perform
determined using estimators (algorithms for performance estimation) > like CBPE and DLE
*estimated when ground truth is not available
realized > represents measured performance
determined using performance calculators (algorithms)
*calculated when ground truth is available

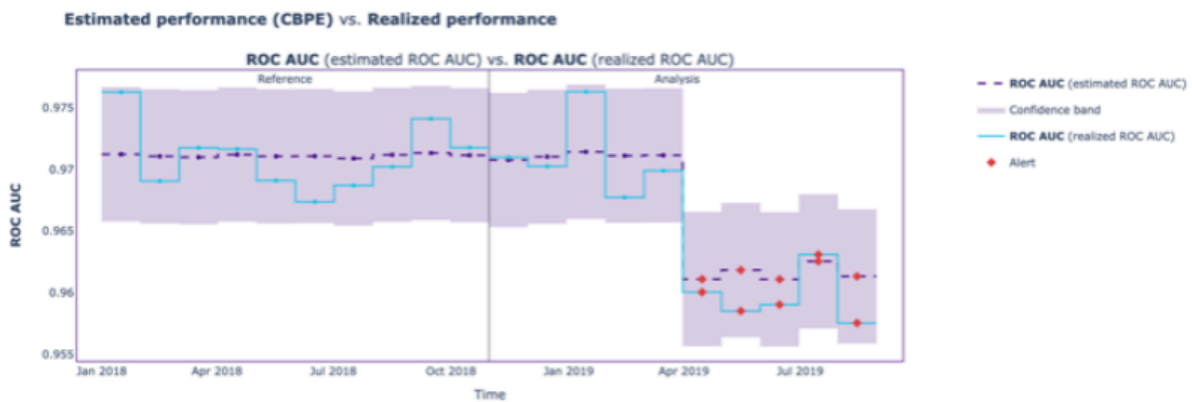
Using NannyML's performance calculator - similar to estimator

```
calculator = nannyml.PerformanceCalculator(
        y_pred_proba = 'y_pred_proba', #this holds the predicted probabilities
        y_pred = 'y_pred', #this holds the model's predicted classes
        y_true = 'targets', #ground truth
        timestamp_column_name = 'timestamp'.,
        metrics = ['roc_auc', 'accuracy'],
        chunk_period = 'd',
        problem_type = 'classification_binary') #this indicates whether we have a
binary or multiclass classification problem
```

#fit the calculator
calc.fit(reference)
realized_results = calc.calculate(analysis)
#**analysis set neeed to include a column with ground truth
results.plot().show()
example output>



Comparing realized and estimated performance
must run the performance estimator and calculator beforehand
using Python:
estimated_results = estimator.estimate(analysis)
realized_results = calculator.calculate(analysis)
#show comparison plot
realized_results.compare(estimated_results).plot().show()
example output>



*in this example the CBPE used is mimicking the model’s actual behavior very well
*if the predictions are far off, it might indicate that concept drift is present in the data

Example

```
# Intialize the calculator
calculator = nannyml.PerformanceCalculator(
    y_true='tip_amount',
    y_pred='y_pred',
    chunk_period='d',
    metrics=['mae'],
    timestamp_column_name='lpep_pickup_datetime',
    problem_type='regression')

# Fit the calculator
calculator.fit(reference)
realized_results = calculator.calculate(analysis)

# Show comparison plot for realized and estimated performance
realized_results.compare(estimated_results).plot().show()
```

How to chunk the data?
time-based, size-based, or number-based
‘chunk’ represents aggregated results for a specific number of observations or a time interval > displayed as a single point on the monitoring plot
size-based > ensures a fixed number of data points per chunk
number-based > specify the total number of chunks we want, ensuring a fixed number of observations per chunk

Initializing custom thresholds
NannyML calculates the mean and standard deviation of the reference data
to compute the lower threshold, it subtracts three standard deviations from the mean
for upper, it adds three standard deviations to the mean
*with NannyML we can also customize this calculation
with Python:

```
from nannyml.thresholds import ConstantThreshold, StandardDeviationThreshold
stdt = StandardDeviationThreshold(
        std_lower_multiplier=3,
        std_upper_multiplier=3)
```

Can also set constant lower and upper thresholds

```
ct = ConstantThreshold(
        lower = 0.85,
        upper = 0.95)
```

*need to pass custom thresholds as a dictionary
with Python:

thresholds={‘roc_auc’’: ct, ‘accuracy’: stdt}

Filtering results
by period >
filtered_results = results.filter(period=‘analysis’)
by metrics >
filtered_results = results.filter(metrics=[‘mae’])
by both
filtered_results = results.filter(period=‘analysis’, metrics=[‘mae’])

We can then export our results to a dataframe
results.filter(period=‘analysis’).to_df()
example output>

	chunk							roc_auc							
	key	chunk_index	start_index	end_index	start_date	end_date	period	value	sampling_error	realized	upper_confidence_boundary	lower_confidence_boundary	upper_threshold	lower_threshold	alert
0	[0:4999]	0	0	4999	None	None	analysis	0.905547	0.002230	NaN	0.912236	0.898859	0.95	0.85	False
1	[5000:9999]	1	5000	9999	None	None	analysis	0.907030	0.002230	NaN	0.913719	0.900342	0.95	0.85	False
2	[10000:14999]	2	10000	14999	None	None	analysis	0.902044	0.002230	NaN	0.908733	0.895355	0.95	0.85	False
3	[15000:19999]	3	15000	19999	None	None	analysis	0.905250	0.002230	NaN	0.911939	0.898562	0.95	0.85	False
4	[20000:24999]	4	20000	24999	None	None	analysis	0.904054	0.002230	NaN	0.910742	0.897365	0.95	0.85	False

Example
reference, analysis, analysis_gt = nannyml.load_us_census_ma_employment_data()

# Initialize the CBPE algorithm
cbpe = nannyml.CBPE(
    y_pred_proba='predicted_probability',
    y_pred='prediction',
    y_true='employed',
    metrics = ['roc_auc', 'accuracy', 'f1'],
    problem_type = 'classification_binary',
    chunk_number = 8,
)

cbpe = cbpe.fit(reference)
estimated_results = cbpe.estimate(analysis)
estimated_results.plot().show()

Example
# Import custom thresholds
from nannyml.thresholds import ConstantThreshold, StandardDeviationThreshold

# Initialize custom thresholds
stdt = StandardDeviationThreshold(std_lower_multiplier = 2, std_upper_multiplier =

2)

```
ct = ConstantThreshold(lower = 0.9, upper = 0.98)

# Initialize the CBPE algorithm
estimator = nannyml.CBPE(
    problem_type='classification_binary',
    y_pred_proba='predicted_probability',
    y_pred='prediction',
    y_true='employed',
    metrics=['roc_auc', 'accuracy', 'f1'],
    thresholds={'f1': ct, 'accuracy': stdt})

# Convert estimated results to a dataframe for the roc_auc metric
display(estimated_results.filter(metrics=['roc_auc']).to_df())

# Convert estimated results to a dataframe for the reference period
display(estimated_results.filter(period='reference', metrics=['f1',
'accuracy']).to_df())

# Show the results plot for the accuracy metric
display(estimated_results.filter(metrics=['accuracy']).plot().show())

# Show the results plot for the analysis set, as well as the accuracy and roc_auc
metrics
display(estimated_results.filter(period='analysis', metrics=['accuracy',
'roc_auc']).plot().show())
```

Business value
NannyML model’s predictions can be organized into a confusion matrix
example hotel bookings:

Here we can use NannyML's 'business value' metric > which weighs the up and down side of TP, FN, FP, and TN

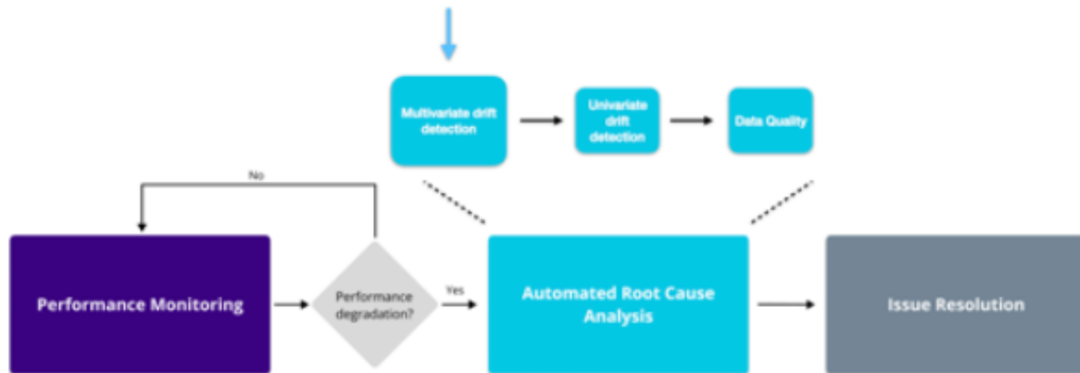
```
# Initialize the calculator
calculator = nannyml.PerformanceCalculator(...
    problem_type='classification_binary',
    metrics=['business_value'],
    # [value_of_TN, value_of_FP], [value_of_FN, value_of_TP]]
    business_value_matrix = [[0, -200],[-100, 1000]],
    normalize_business_value='None')
```

parameter normalize_business_value can be either 'None' or 'per_prediction' determines whether the results are shown for the entire chunk or each prediction
*we use 'per_prediction' when info (this case booking cancellations) are not available

Example

```
# Custom business value thresholds
ct = ConstantThreshold(lower=0, upper=150000)
# Intialize the performance calculator
calc = PerformanceCalculator(problem_type='classification_binary',
          y_pred_proba='y_pred_proba',
          timestamp_column_name="timestamp",
          y_pred='y_pred',
          y_true='is_canceled',
          chunk_period='m',
          metrics=['business_value', 'roc_auc'],
          business_value_matrix = [[0, -100],[-200, 1500]],
          thresholds={'business_value': ct})
calc = calc.fit(reference)
calc_res = calc.calculate(analysis)
calc_res.filter(period='analysis').plot().show()
```

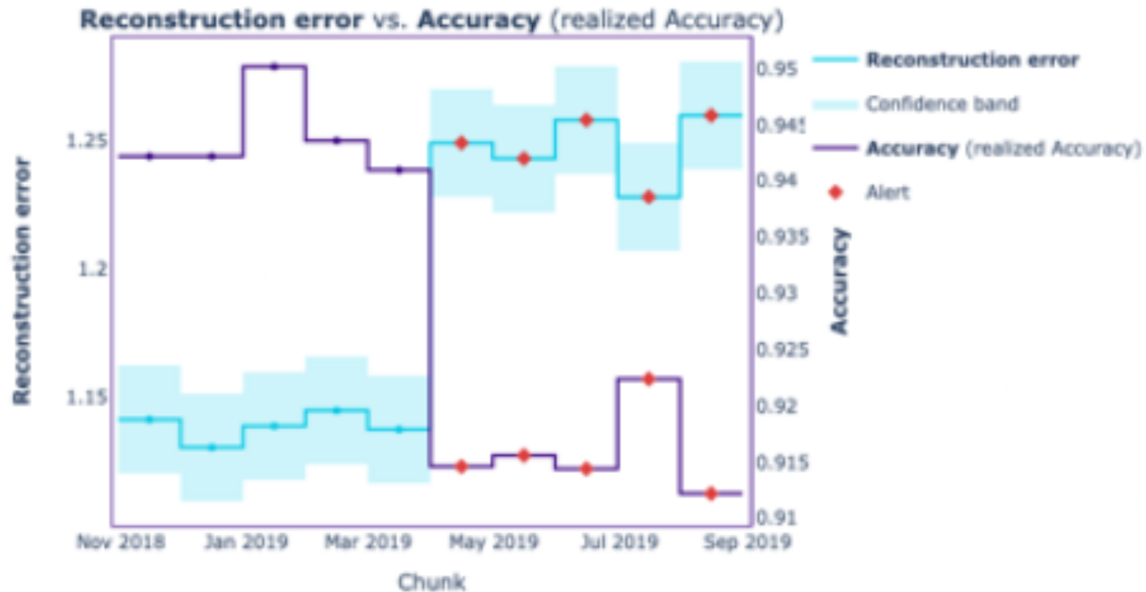
Multivariate drift detection > first step of RCA



How multivariate drift detection works?
we use the PCA algorithm to compress the data > giving us latent space data
then we decompress the data with an inverse PCA algorithm
*then we measure the reconstruction error > increase indicates data drift
NannyML calculates this error for each chunk and raises an alert when the values get outside of the thresholds defined in the reference period
with Python:

```
mv_calc = nannyml.DataReconstructionDriftCalculator(...
#fit
mycalc.fit(reference)
mv_results = mv_calc.calculate(analysis)
figure = mv_results.filter(period='analysis').compare(perf_results).plot()
figure.show()
```

example comparison graph>



Example

```
# Create standard deviation thresholds
stdt = StandardDeviationThreshold(std_lower_multiplier=2, std_upper_multiplier=1)

# Define feature columns
feature_column_names = ['country', 'lead_time', 'parking_spaces', 'hotel']

# Intialize, fit, and show results of multivariate drift calculator
mv_calc = nannyml.DataReconstructionDriftCalculator(
    column_names=feature_column_names,
    threshold = stdt,
    timestamp_column_name='timestamp',
    chunk_period='m')
mv_calc.fit(reference)
mv_results = mv_calc.calculate(analysis)
mv_results.filter(period='analysis').compare(perf_results).plot().show()
```

Univariate drift detection
*method used after the multivariate one
look at each feature individually to determine why and if it is drifting
result is a single number which represents the amount of drift between the reference and analysis chunk
NannyML supports six methods:

Univariate methods

- Jensen-Shannen distance - both categorical and continuous
- Hellinger - categorical and continuous
- Wasserstein - only continuous
- Kolgomorov-Smirnov - only continuous
- L-infinity - only categorical
- Chi2 - only categorical

Using Python:

```
# Intialize the univariate drift calculator
uv_calc = nannyml.UnivariateDriftCalculator(
    continuous_methods=['wasserstein', 'hellinger'],
    categorical_methods=['jensen_shannon', 'l_infinity', 'chi2'],
    column_names=feature_column_names,
    timestamp_column_name='timestamp',
    chunk_period='d'
    )
```

```
# Fit, calculate and plot the results
uv_calc.fit(reference)
uv_results = uv_calc.calculate(analysis)
uv_results.plot().show()
```

we are able to filter by column names and methods
filtered_figure = uv_results.filter(column_names=['trip_distance', 'fare_amount'],
    methods=['jensen_shannon'])
filtered_figure.show().plot()
**if too many features we can NannyML's ranker

Two rankers
Alert counting > rank features based on the number of alerts

```
# Initialize the alert count ranker
alert_count_ranker = nannyml.AlertCountRanker()
alert_count_ranked_results = alert_count_ranker.rank(
    uv_results,
    only_drifting=False)
# Display the results
display(alert_count_ranked_results)
```

	number_of_alerts	column_name	rank
0	4	DOLocationID	1
1	3	fare_amount	2
2	1	trip_distance	3
3	1	PULocationID	4

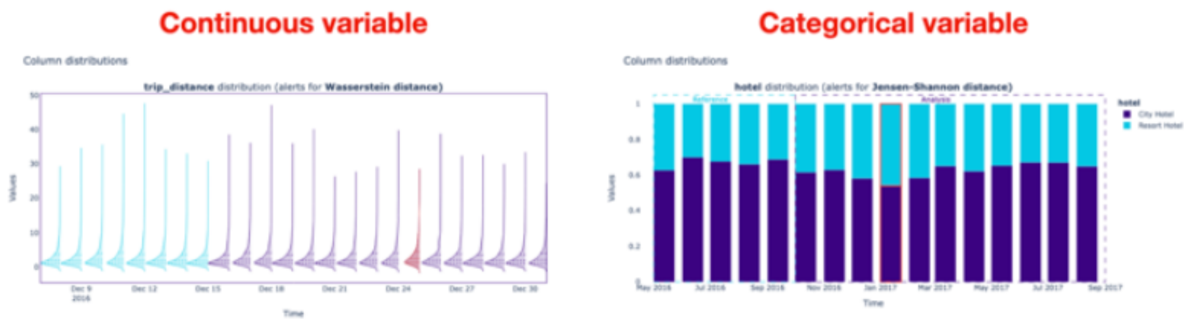
Considering that many alerts may be false, we can use the correlation ranker to validate them
Correlation ranker > ranks features based on how much they correlate to absolute changes in performance

```
# Initialize the correlation ranker
correlation_ranker = nannyml.CorrelationRanker()
correlation_ranker.fit(perf_results.filter(period='reference'))
correlation_ranked_results = correlation_ranker.rank(uv_results, perf_results)

# Display the results
display(correlation_ranked_results)
```

	column_name	pearsonr_correlation	pearsonr_pvalue	has_drifted	rank
0	trip_distance	0.736320	0.000041	True	1
1	DOLocationID	0.257138	0.225134	True	2
2	fare_amount	0.193746	0.364340	True	3
3	PULocationID	-0.071132	0.741181	True	4

NannyML can track how the feature distributions evolve in each chunk
this can significantly improve our understanding of drift and its connection to performance
To use:
#create distribution plots

```
distribution_results = uv_results.plot(kind=‘distribution’)
distribution_results.show()
```
example outputs>

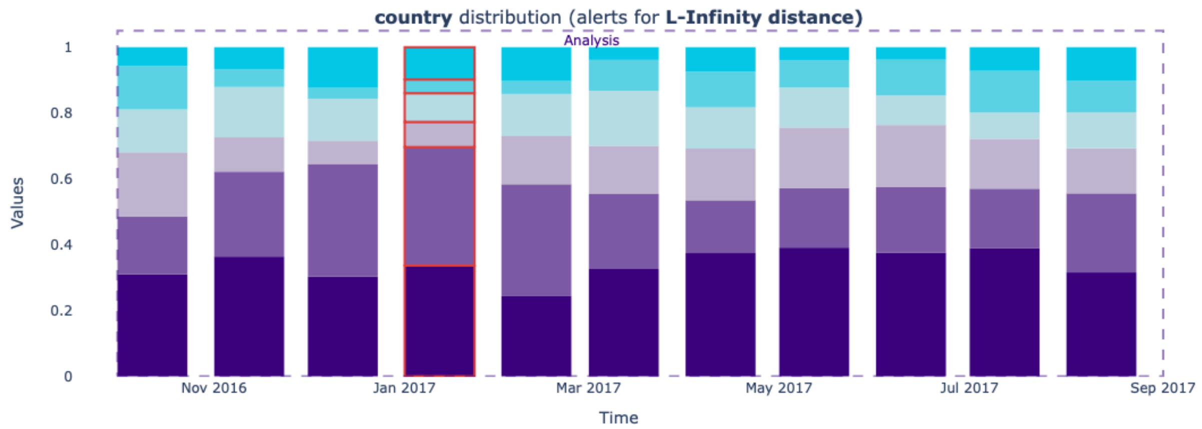


Example

```
# Initialize the alert count ranker
alert_count_ranker = nannyml.AlertCountRanker()
alert_count_ranked_features = alert_count_ranker.rank(
    uv_results.filter(methods=['wasserstein', 'l_infinity']))

display(alert_count_ranked_features)

# Initialize the correlation ranker
correlation_ranker = nannyml.CorrelationRanker()
correlation_ranker.fit(perf_results.filter(period='reference'))

correlation_ranked_features = correlation_ranker.rank(
    uv_results.filter(methods=['wasserstein', 'l_infinity']),
    perf_results.filter(methods=['wasserstein', 'l_infinity']))
display(correlation_ranked_features)

# Filter and create drift plots
drift_results = uv_results.filter(
    period='analysis',
    column_names=['hotel', 'country']
    ).plot(kind='drift')

# Filter and create distribution plots
distribution_results = uv_results.filter(
    period='analysis',
    column_names=['hotel', 'country']
    ).plot(kind='distribution')

# Show the plots
```

drift_results.show()
distribution_results.show()

output>



Data quality and statistic checks
-missing value detection
-unseen value detection
monitor row count for each chunk > if too low, it might not be enough data to calculate univariate or multivariate results

Missing values detection

```
# Instantiate the missing values calculator module
ms_calc = nannyml.MissingValuesCalculator(column_names=["Age"], normalize=True)

# Fit the calculator on the reference set
ms_calc.fit(reference)

# Calculate the rate of the missing values on the analysis set
ms_results = ms_calc.calculate(analysis)
ms_results.plot()
```

set normalize parameter to True if you want to see the ratio of missing values

Unseen values detection
categorical feature values that are not present in the reference period

```
# Instantiate the unseen values calculator module
us_calc = nannyml.UnseenValuesCalculator(column_names=["Cabin"], normalize=False)
# Fit, calculate and plot the rate of the unseen values
us_calc.fit(reference)
us_results = us_calc.calculate(analysis)
us_results.plot()
```

Data quality check with summary statistics

- **Summation:** Useful for financial data to calculate revenue, or profits for a specific period.
- **Mean** and **Standard Deviation:** Helpful for data drift check and explainability.
- **Median:** Resistant to outliers, making it useful when dealing with features that have many extreme values.
- **Row Counts:** Determine if there is enough data in each chunk.

```
sum_calc = nannyml.SummaryStatsSumCalculator(column_names=selected_columns)
avg_calc = nannyml.SummaryStatsAvgCalculator(column_names=selected_columns)
std_calc = nannyml.SummaryStatsStdCalculator(column_names=selected_columns)
med_calc = nannyml.SummaryStatsMedianCalculator(column_names=selected_columns)
rows_calc = nannyml.SummaryStatsRowCountCalculator(column_names=selected_columns)
```

Example

```
# Define analyzed columns
selected_columns = ['country', 'lead_time', 'parking_spaces', 'hotel']

# Intialize missing values calculator
ms_calc = nannyml.MissingValuesCalculator(
    column_names=selected_columns,
    chunk_period='m',
    timestamp_column_name='timestamp'
)

# Fit, calculate and plot the results
ms_calc.fit(reference)
ms_results = ms_calc.calculate(analysis)
ms_results.plot().show()

# Define analyzed column
analyzed_column = ['lead_time']
```

```
# Intialize median values calculator
med_calc = nannyml.SummaryStatsMedianCalculator(
    column_names=analyzed_column,
    chunk_period='m',
    timestamp_column_name='timestamp'
)

# Fit, calculate and plot the results
med_calc.fit(reference)
med_calc_res = med_calc.calculate(analysis)
med_calc_res.filter(period='analysis').plot().show()
```

Issue resolution

- do nothing
- retrain
  - on both old and new data
  - fine-tune the old model with the new data
  - weighting data > give more importance to the recent data if the new data is more relevant to the business problem
- revert back to a previous model
- go downstream of the model and business process (ie having a branch manager use experience to order weekly toilet paper need if model is underperforming