Comparison of Riemannian Classifiers

Riemannian geometry offers methods for extracting features for classification from multichannel EEG signals, by treating functional connectivity matrices as points on a Riemannian manifold - a smooth, curved space where distances between points are defined by the geometry of the manifold rather than the linear geometry of Euclidean space. We demonstrate how to estimate pairwise connectivity matrices between EEG channels in both the time (covariance) and frequency (cross-spectral density) domains. Importantly, both forms are symmetric/hermitian positive definite and reside on a Riemannian manifold. Using Riemannian distances between matrices, we compare a number of classification algorithms from the pyRiemann package.

In [1]:
```
import numpy as np
from aeon.datasets import load_classification
from aeon.transformations.collection.base import BaseCollectionTransformer
from pyriemann.classification import (
    MDM,
    SVC,
    FgMDM,
    KNearestNeighbor,
    MeanField,
    TSclassifier,
)
from sklearn.pipeline import make_pipeline

from aeon_neuro._wip.transformations.series._covariance import CovarianceMatrix
from aeon_neuro._wip.transformations.series._power_spectrum import CrossSpectralMatrix
```

```
/Users/griegner/git-repositories/aeon-neuro/.venv/lib/python3.12/site-packages/aeon/base/__init__.py:24: FutureWarning: The ae
on package will soon be releasing v1.0.0 with the removal of legacy modules and interfaces such as BaseTransformer and BaseFor
ecaster. This will contain breaking changes. See aeon-toolkit.org for more information. Set aeon.AEON_DEPRECATION_WARNING or t
he AEON_DEPRECATION_WARNING environmental variable to 'False' to disable this warning.
  warnings.warn(
OMP: Info #276: omp_set_nested routine deprecated, please use omp_set_max_active_levels instead.
```

Load EEG Classification Dataset

The KDD dataset segments an EEG series from a single subject into a collection of epochs with corresponding labels indicating whether the subject's hand is resting on a table ("rest") or raised ("task"). Some processing steps have been pre-applied: bandpass filtering between 0.5Hz-100Hz, downsampling to 100 timepoints, and applying a channel selection algorithm (see the data loading example). Both the training and testing sets have shape '(40_epochs, 4_channels, 100_timepoints)' with 20 epochs of each class.

In [ ]:
```
X_train, y_train = load_classification(name="KDD_MTSC", split="TRAIN", extract_path="../aeon_neuro/data/KDD_Examples")

X_test, y_test = load_classification(name="KDD_MTSC", split="TEST", extract_path="../aeon_neuro/data/KDD_Examples")

print(f"X shape: {X_train.shape}\ty shape: {y_train.shape}")
```

```
X shape: (40, 4, 100)   y shape: (40,)
```

Pipeline of Transformers and Classifiers

The transformers take a collection of multivariate series of '(n_epochs, n_channels, n_timepoints)' and output functional connectivity matrices for each epoch, '(n_epochs, n_channels, n_channels)'. Specifically, we estimate covariance and cross-spectral density matrices. Since some of the classifiers used are only defined on real-valued domains, so we compute the magnitude of the cross-spectral densities, which takes the absolute value of the real part of the complex entries of the matrix. The pipeline first applies these transformers, then classifies the resulting matrices using Riemannian means and distances between matrices. The following classifiers are compared (see pyRiemann documentation):

1. minimum distance to mean
2. minimum distance to mean with geodesic smoothing
3. tangent space embedding
4. k-nearest neighbors
5. support-vector machine
6. minimum distance to mean field

In [ ]:
```
class SeriesToCollectionWrapper(BaseCollectionTransformer):
    """Treat a SeriesTransformer as a CollectionTransformer.

    Parameters
    ----------
    transformer : SeriesTransformer
        The transformer to wrap.
    """

    _tags = {"capability:multivariate": True}

    def __init__(self, transformer):
        self.transformer = transformer
        super().__init__()

    def _transform(self, X, y=None):
        return np.array([self.transformer.fit_transform(x) for x in X])


# initialize transformers and classifiers
# use magnitude of the CSD bc many of the classifiers handle real-valued matrices
cov_transformer = SeriesToCollectionWrapper(CovarianceMatrix())
csd_transformer = SeriesToCollectionWrapper(CrossSpectralMatrix(sfreq=112.5, magnitude=True))

# all classifiers compare SPD matrices by a Riemannian distance measure
classifiers = {
    "minimum distance to mean": MDM(),
    "geodesic filtering": FgMDM(),
    "tangent space embedding": TSclassifier(),
    "k-nearest neighbors": KNearestNeighbor(),
    "support-vector machine": SVC(),
    "minimum distance to mean field": MeanField(),
}

pipelines = {}
for name, clf in classifiers.items():
    pipelines[f"cov matrix + {name}"] = make_pipeline(cov_transformer, clf)
    pipelines[f"csd matrix + {name}"] = make_pipeline(csd_transformer, clf)
```

Compare Classification Accuracies

In [4]:
```
for name, pipeline in pipelines.items():
    pipeline.fit(X_train, y_train)
    y_pred = pipeline.predict(X_test)
    print(f"{name: <45} accuracy: {np.mean(y_pred == y_test)}")
```

```
cov matrix + minimum distance to mean         accuracy: 0.675
csd matrix + minimum distance to mean         accuracy: 0.8
cov matrix + geodesic filtering               accuracy: 0.8
csd matrix + geodesic filtering               accuracy: 0.875
cov matrix + tangent space embedding          accuracy: 0.7
csd matrix + tangent space embedding          accuracy: 0.825
cov matrix + k-nearest neighbors              accuracy: 0.875
csd matrix + k-nearest neighbors              accuracy: 0.925
cov matrix + support-vector machine           accuracy: 0.7
csd matrix + support-vector machine           accuracy: 0.85
cov matrix + minimum distance to mean field   accuracy: 0.675
csd matrix + minimum distance to mean field   accuracy: 0.725
```