

# Build in models

## PCA

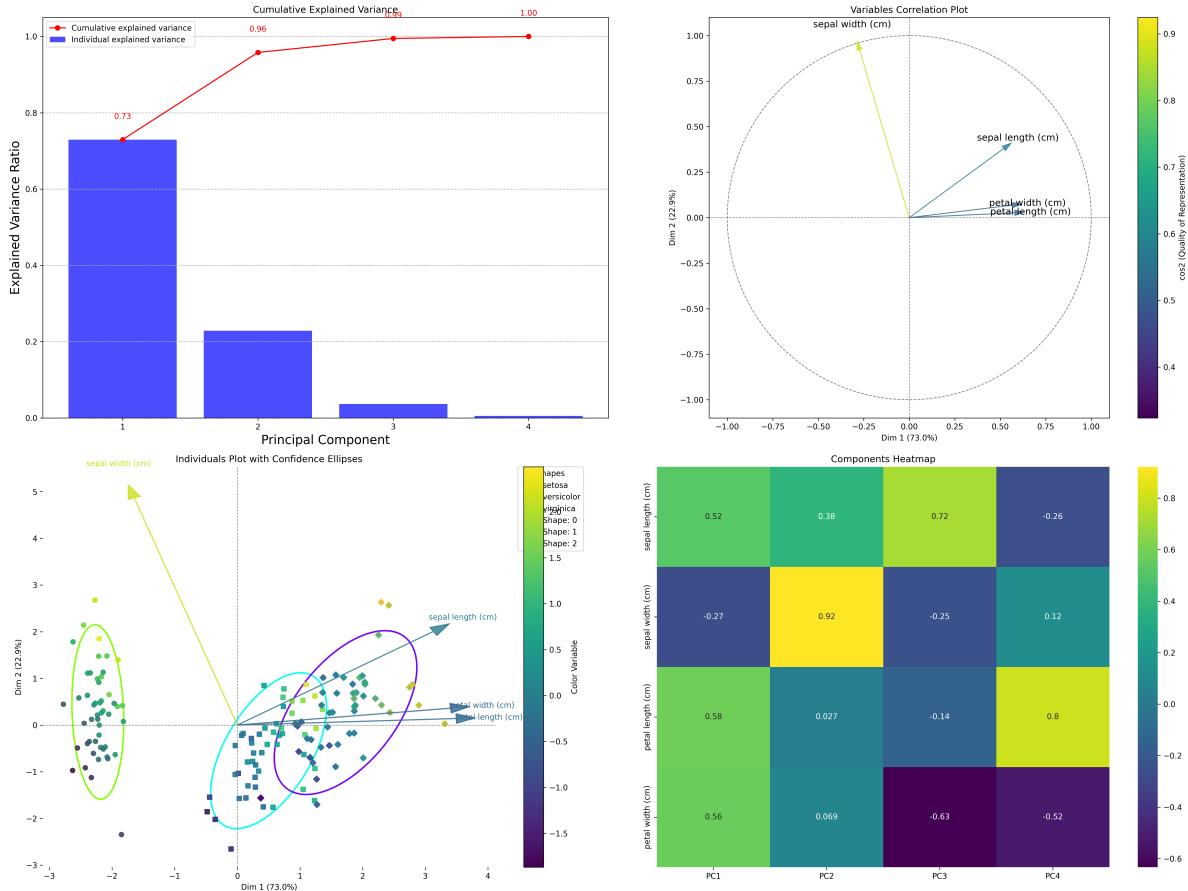
```
from BI import bi, jnp

m=bi()
m.data('iris.csv', sep=',') # Data is already scaled
m.data_on_model = dict(
    X=jnp.array(m.df.iloc[:,0:-2].values)
)
m.fit(m.models.pca(type="classic"), progress_bar=False) # or robust, sparse, classic, sparse

m.models.pca.plot(
    X=m.df.iloc[:,0:-2].values,
    y=m.df.iloc[:, -2].values,
    feature_names=m.df.columns[0:-2],
    target_names=m.df.iloc[:, -1].unique(),
    color_var=m.df.iloc[:, 0].values,
    shape_var=m.df.iloc[:, -2].values
)

jax.local_device_count 16
```

## PCA Analysis Dashboard



## Survival analysis

```
from BI import bi, jnp

m = bi()
m.data('mastectomy.csv', sep=',').head()
m.df.metastasized = (m.df.metastasized.values == "yes").astype(jnp.int64)

# Import time-steps and events
m.models.survival.import_time_event(
    m.df.time.values,
    m.df.event.values, interval_length=3
```

```

)
# To import time-fixed covariates
m.models.survival.import_covF(
    m.df.metastasized.values, ['metastasized']
)

# To import time-varying covariates Experimental feature
# m.models.survival.import_covV

m.fit(m.models.survival.model, num_samples=500)

m.summary()

m.models.survival.plot_surv( beta = 'Hazard_rate_metastasized')

```

jax.local\_device\_count 16

---

Survival concern 44 individuals in 76 intervals.  
18.0 individuals experienced the event.

---

Covariates imported: ['metastasized']  
Surv object now has 1 covariates: ['metastasized']  
This function is still in development. Use it with caution.  
This function is still in development. Use it with caution.

0% | 0/1000 [00:00<?, ?it/s]

This function is still in development. Use it with caution.

warmup: 0% | 1/1000 [00:02<46:26, 2.79s/it, 1 steps of size 2.34e+00. acc. prob-

This function is still in development. Use it with caution.  
This function is still in development. Use it with caution.  
This function is still in development. Use it with caution.

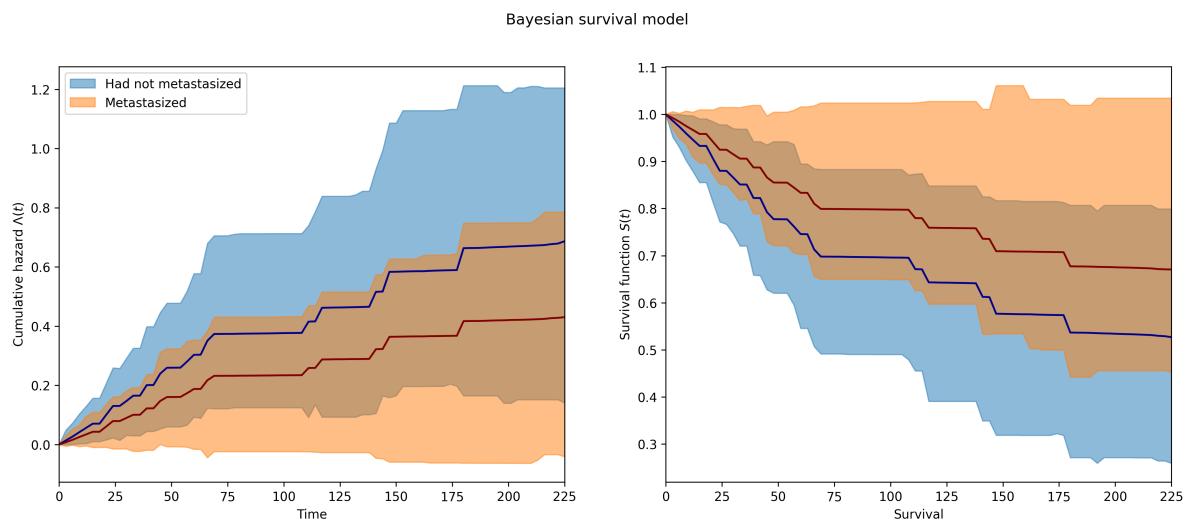
arviz - WARNING - Shape validation failed: input\_shape: (1, 500), minimum\_shape: (chains=2, c  
/home/sosa/work/.venv/lib/python3.12/site-packages/arviz/plots/hdiplot.py:166: FutureWarning

hdi currently interprets 2d data as (draw, shape) but this will change in a future release to

```

/home/sosa/work/.venv/lib/python3.12/site-packages/arviz/plots/hdiplot.py:166: FutureWarning
    hdi currently interprets 2d data as (draw, shape) but this will change in a future release to
/home/sosa/work/.venv/lib/python3.12/site-packages/arviz/plots/hdiplot.py:166: FutureWarning
    hdi currently interprets 2d data as (draw, shape) but this will change in a future release to
/home/sosa/work/.venv/lib/python3.12/site-packages/arviz/plots/hdiplot.py:166: FutureWarning
    hdi currently interprets 2d data as (draw, shape) but this will change in a future release to

```



## Gaussian Mixture Models

```

from BI import bi
from sklearn.datasets import make_blobs
m = bi()

# Generate synthetic data
data, true_labels = make_blobs(
    n_samples=500, centers=8, cluster_std=0.8,
    center_box=(-10,10), random_state=101
)

```

```
m.data_on_model = {"data": data,"K": 8 }
m.fit(m.models.gmm) # Optimize model parameters through MCMC sampling
m.plot(X=data,sampler=m.sampler) # Experimental feature
```

```
jax.local_device_count 16
```

```
0%|          | 0/1000 [00:00<?, ?it/s] warmup: 0%|          | 1/1000 [00:02<43:32, 2.61s]
```

## Dirichlet Process Mixture Models

### Python

```
from BI import bi
from sklearn.datasets import make_blobs
m = bi()

# Generate synthetic data
data, true_labels = make_blobs(
    n_samples=500, centers=8, cluster_std=0.8,
    center_box=(-10,10), random_state=101
)
m.data_on_model = dict(data=data,T=10)
m.fit(m.models.dpmm)
m.plot(data,m.sampler) # Experimental feature
```

```
jax.local_device_count 16
```

```
0%|          | 0/1000 [00:00<?, ?it/s] warmup: 0%|          | 1/1000 [00:02<46:26, 2.79s]
```

## Network Models

### Python

```

# Setup device-----
from BI import bi, jnp

# Setup device-----
m = bi(platform='cpu')
# Simulate data -----
N = 50
individual_predictor = m.dist.normal(0,1, shape = (N,1), sample = True)

kinship = m.dist.bernoulli(0.3, shape = (N,N), sample = True)
kinship = kinship.at[jnp.diag_indices(N)].set(0)

def sim_network(kinship, individual_predictor):
    # Intercept
    alpha = m.dist.normal(0,1, sample = True)

    # SR
    sr = m.net.sender_receiver(individual_predictor, individual_predictor, s_mu = 0.4, r_mu = 0.4)

    # D
    DR = m.net.dyadic_effect(kinship, d_sd=2.5, sample = True)

    return m.dist.bernoulli(logits = alpha + sr + DR, sample = True)

network = sim_network(m.net.mat_to_edgl(kinship), individual_predictor)

# Predictive model -----
m.data_on_model = dict(
    network = network,
    dyadic_predictors = m.net.mat_to_edgl(kinship),
    focal_individual_predictors = individual_predictor,
    target_individual_predictors = individual_predictor
)

def model(network, dyadic_predictors, focal_individual_predictors, target_individual_predictors):
    N_id = network.shape[0]

    # Block -----
    alpha = m.dist.normal(0,1, sample = True)

```

```
## SR shape = N individuals-----
sr = m.net.sender_receiver(
  focal_individual_predictors,
  target_individual_predictors,
  s_mu = 0.4, r_mu = -0.4
)

# Dyadic shape = N dyads-----
dr = m.net.dyadic_effect(dyadic_predictors, d_sd=2.5) # Diadic effect intercept only

m.dist.bernoulli(logits = alpha + sr + dr, obs=network)

m.fit(model, num_samples = 500, num_warmup = 500, num_chains = 1, thinning = 1)
```