Principal Component Analysis (PCA)

General Principles

Principal Component Analysis (PCA) is a technique used to reduce the dimensionality of a dataset by transforming it into a new coordinate system where the greatest variance of the data is projected onto the first coordinates (called principal components). This method helps capture the underlying structure of high-dimensional data by identifying patterns based on variance.

In **Bayesian PCA**, uncertainty in the model parameters is explicitly taken into account by using a probabilistic framework. This allows us to not only estimate the principal components but also quantify the uncertainty around them and avoid overfitting by incorporating prior knowledge.

Goal:

- **Reduce dimensionality** while retaining as much variance as possible.
- **Infer posterior distributions** over the principal components, instead of point estimates, by incorporating prior distributions over the parameters.

Use Cases

- **Dimensionality Reduction**: Bayesian PCA is commonly used to reduce the dimensionality of high-dimensional datasets while incorporating uncertainty about the latent structure.
- **Data Visualization**: By projecting data into a lower-dimensional space, PCA helps in visualizing high-dimensional datasets in 2D or 3D plots.
- **Noise Modeling**: Bayesian PCA provides an advantage over classical PCA by explicitly modeling noise and accounting for uncertainty in the data.

- **Feature Extraction**: The latent variables learned by Bayesian PCA can serve as new features for downstream tasks, such as classification or clustering.
- **Latent Variable Modeling**: The latent variables learned by Bayesian PCA can serve as new features for downstream tasks, such as classification or clustering.

Considerations

In **Bayesian PCA**, we assume prior distributions for the latent variables Z and the principal component loadings W. We place Gaussian priors on both Z and W and learn their posterior distributions using the observed data X.

This approach differs from traditional PCA by allowing the posterior distributions to reflect uncertainty in the model parameters.

Example

Here is an example code snippet demonstrating Bayesian PCA using TensorFlow Probability:

```
from main import *
import seaborn as sns

m = bi(platform='cpu')

# Data simulation ----------------------------------------

plt.style.use("ggplot")
warnings.filterwarnings('ignore')

num_datapoints = 5000
data_dim = 2
latent_dim = 1
stddv_datapoints = 0.5

# Simulate data
def sim_data(data_dim, latent_dim, num_datapoints, stddv_datapoints, seed = 0):
    w = bi.dist.normal(0, 1, shape=(data_dim, latent_dim), name='w', sample=True, seed=seed)
    z = bi.dist.normal(0, 1, shape=(latent_dim, num_datapoints), name='z', sample=True, seed=
    x = bi.dist.normal(w @ z, stddv_datapoints, name='x', sample=True, seed=seed)
    return w, z, x

actual_w, actual_z, x_train =sim_data(data_dim, latent_dim, num_datapoints, stddv_datapoints
```

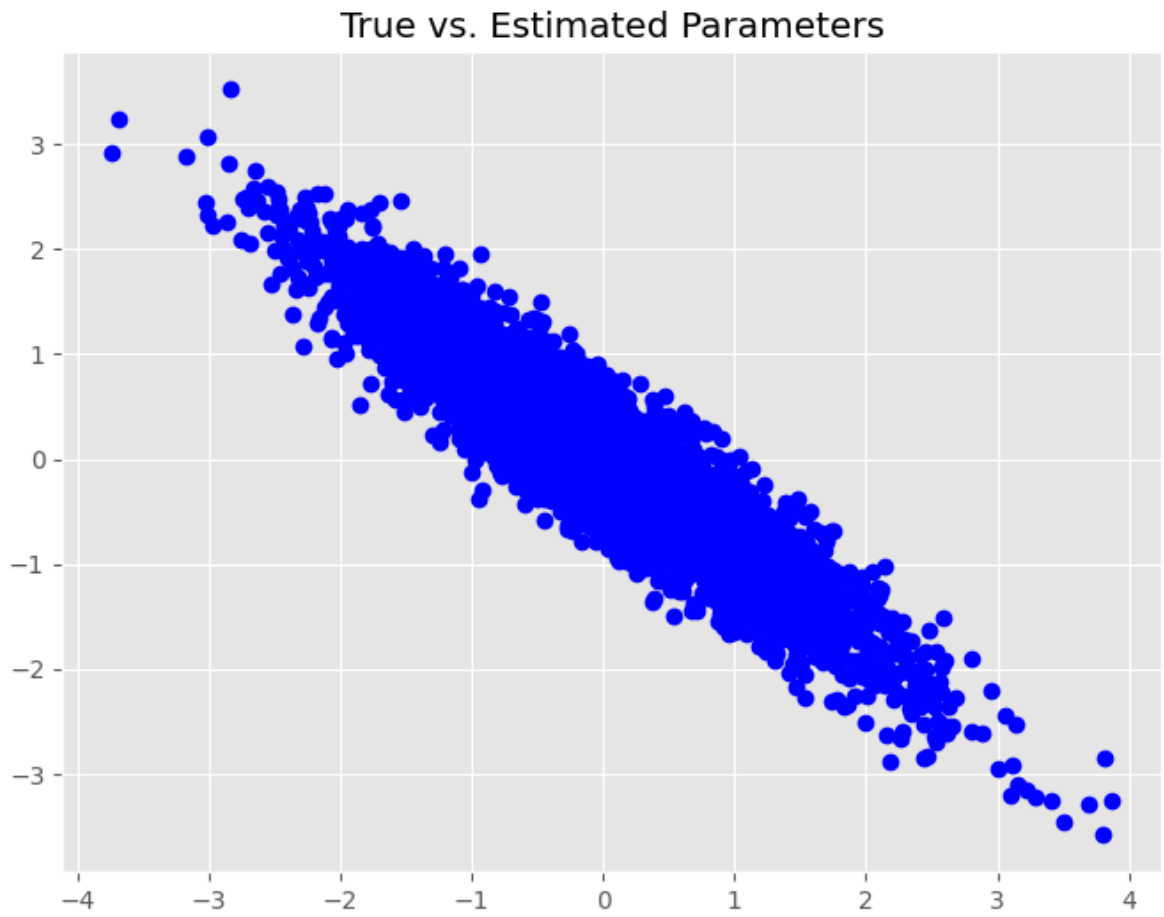
```
plt.scatter(x_train[0, :], x_train[1, :], color='blue', alpha=0.1)
plt.axis([-20, 20, -20, 20])
plt.title("Dataset")
plt.show()


# Model using simulated data
def model(x_train, data_dim, latent_dim, num_datapoints, stddv_datapoints, seed = 0):
    w = bi.dist.normal(0, 1, shape=(data_dim, latent_dim), name='w')
    z = bi.dist.normal(0, 1, shape=(latent_dim, num_datapoints), name='z')
    lk('Y', Normal(w @ z, stddv_datapoints), obs = x_train)

m.data_on_model = dict(
    x_train = x_train,
    data_dim = data_dim,
    latent_dim = latent_dim,
    num_datapoints = num_datapoints,
    stddv_datapoints = stddv_datapoints
)

m.fit(model)
summary = m.summary()
real_data = jnp.concatenate([actual_w.flatten(), actual_z.flatten()]) # concatenate the actua
posteriors = summary.iloc[:,0]


plt.figure(figsize=(8, 6))
plt.plot(real_data, posteriors, marker='o', linestyle='None', color='b', label='Posteriors')
```



Mathematical Details

Formulation

Given an observed data matrix $X \in \mathbb{R}^{N \times D}$ (where N is the number of samples and D is the number of dimensions), we assume the data is generated by a lower-dimensional latent variable model:

$$X = ZW^T + \epsilon$$

$$Z \sim \mathcal{N}(0, I)$$

$$W \sim \mathcal{N}(0, I)$$

$$\epsilon \sim \mathcal{N}(0, \sigma^2 I)$$

Where:

- X is the observed data matrix.
- $Z \in \mathbb{R}^{N \times K}$ is the latent variable matrix (latent features with $K \ll D$). Z is defined by a Normal distribution with mean 0 and variance 1.
- $W \in \mathbb{R}^{D \times K}$ is the matrix of principal components (*projection matrix*). W is defined by a Normal distribution with mean 0 and variance 1.
- ϵ is Gaussian noise, assumed to be normally distributed: $\epsilon \sim \mathcal{N}(0, \sigma^2 I)$.

Note

- To account for **sign ambiguity** in PCA, we can align the signs of the estimated parameters with the true parameters before comparison. To do this, calculate the dot product between the true and estimated parameters. If it is negative, multiply the estimated parameters by -1 to align them with the true parameters. Below, a code snippet highlights how to do this:

```
true_params = jnp.array(real_data)
estimated_params = jnp.array(posteriors)

# Compute dot product
dot_product = jnp.dot(true_params, estimated_params)

# Align signs if necessary
if dot_product < 0:
    estimated_params = -estimated_params

# Plot the aligned parameters
plt.scatter(true_params, estimated_params, alpha=0.7)
plt.plot([min(true_params), max(true_params)], [min(true_params), max(true_params)], 'r--')
plt.xlabel('True Parameters')
plt.ylabel('Estimated Parameters')
plt.title('True vs. Estimated Parameters After Sign Alignment')
plt.show()
```

Reference(s)

https://www.tensorflow.org/probability/examples/Probabilistic_PCA