P1 ICA Implementation

William Koehrsen wjk68

April 25, 2018

Contents

1 Introduction: FastICA Implementation

In this notebook, we look at implementing independent component analysis (ICA). This method can be used for blind source separation, or in other words, to determine the independent individual signals comprising a single mixed signal. Applications of ICA include separating out the individual parts of audio signals or dimensionality reduction for classification/regression machine learning models. For example, if there are 3 instruments playing at the same time, ICA can be used to separate out the individual instruments from the composite signal. The components returned by ICA are maximally independent, which means that information on one component does not give information on the other components. ICA operates under the assumption that the components are independent and non-Gaussian.

```
In [1]: # Pandas and numpy for data manipulation
        import pandas as pd
        import numpy as np

In [2]: # Utilities for linear algebra
        from scipy import linalg

        # Convergence check
        import warnings
```

2 Implementation of FastICA

The particular implementation of Independent Component Analysis we will use here is known as fast ica. Reference for this method can be found in the paper "Independent Component Analysis: Algorithms and Applications" by Aapo Hyvärinen and Erkki Oja.

Additional information on ICA used in the report can be found in this book. In particular, the algorithm is developed using information presented in Chapter 8 (pages 165-199).

A third source consulted for this method was the paper "Independent Component Analysis by Minimization of Mutual Information" by Aapo Hyvärinen.

ICA assumes the data matrix, X, to be a linear combination of non-Gaussian (independent) components. In other words, $X = \text{mixing} * \text{source}$ where mixing and source are both matrices (this equation is generally written as $X = AS$). The columns of the source matrix contain the independent components, and A is the linear mixing matrix. The objective of ICA is to un-mix the data by finding the un-mixing matrix (W). The source matrix, S, can then be found from $S = WKX$ where K is a pre-whitening matrix that projects the data matrix onto the principal components.

When running ICA, we need to specify a number of components, `n_components`. The shape of the matrices are:

- X, data matrix: [`n_samples, n_features`]
- S, source matrix: [`n_samples, n_components`]
- W, un-mixing matrix: [`n_components, n_features`]
- K, whitening matrix: [`n_components, n_features`]

Once the S, W, and K matrices have been found, the mixing matrix can be calculated from $A = WK^{-}1$ where the shape is [`n_features, n_components`].

2.1 Utility Function: Negentropy

The cost function used in ICA is the negative entropy, negentropy. Negative entropy is a measure of non-Gaussianity that is based on the information-theroy quantity of entropy. The more unpredictable a variable, the greater its entropy. A Gaussian random variable has maximum entropy. Rather than using entropy, we will use a measure called negentropy, which is always non-negative and zero if and only if the variable has a Gaussian

distribution. Therefore, if the utility function is Negentropy, the goal is to maximize this quantity. The definition of Negentropy is:

$$J(y) = H(y_{gauss}) - H(y)$$

where y_{gauss} is a random Gaussian varialbe with the same covariance matrix as y.
This follows from the definition of Entropy for a continuous variable:

$$H(y) = -\int f(y) \log f(y) d(y)$$

Negentropy is extremely computationally expensive to calculate because it requires estimation of the probability density function (pdf). Therefore approximations of Negentropy are used in practice. In this notebook we will be using the following approximation.

$$J(y) \approx \sum_{i=1}^{p} k_i [E\{G_i(y)\} - E\{G_i(v)\}]^2,$$

Where v is a Guassian variable of zero mean and unit variance, k_i are positive constants, the variable y has zero mean and unit variance, and G_i are nonquadratic functions. We will use a single non-quadratic function. Good choices for G are functions that do not grow too fast.

The following is the log cosh function that we will be using for the non-quadratic function:

$$G_1(u) = \frac{1}{a_1} \log \cosh a_1 u$$

Here $1 \leq a_1 \leq 2$ and the overall method gives a good approximation of Negentropy.
We can use the identity

$$\frac{d}{dx}(\log \cosh x) = \tanh x$$

In [3]:
```
# Calculate g and g_ for a function
# Standard non_linear function
def logcosh(x, alpha = 1.0):

    x *= alpha

    # Apply tanh inplace
    # Using the identity that the deri
    gx = np.tanh(x, x)
    g_x = np.empty(x.shape[0])

    for i, gx_i in enumerate(gx):
        g_x[i] = (alpha * (1 - gx_i ** 2)).mean()

    return gx, g_x
```

3 Preprocessing

3.1 Centering

This simply entails subtracting the mean from each feature of a sample. The resulting variable is a zero-mean variable. The mean can be added back into the centered sources after the mixing matrix has been estimated with the centered data. Also, to re-construct the original signal, we need to add the mean back into the product of the mixing and source matrices.

3.2 Whitening Components

The observed variables must be whitened after they are centered. Whitening results in a new vector that has uncorrelated coponents with variances equal to unity. The result is that the covariance matix of the vector quals the identity matrix.

Here we will use the eigenvalue decomposition of the convariance matrix to whiten the components. The whitening process reudcues the number of parameters that need to be estaimed because the new mixing matrix is orthogonal. This reduces the degrees of freedom and reduces the complexity of the independent component analysis problem.

```
In [4]: # Center and Whiten components
        def whiten_components(X, n_components):

            n, p = X.shape

            X_mean = X.mean(axis=-1)

            # Subtract the mean for 0 mean
            X -= X_mean[:, np.newaxis]

            # Preprocessing by PCA
            u, d, _ = linalg.svd(X, full_matrices=False)

            # Whitening matrix
            whitening = (u / d).T[:n_components]

            # Project data onto the principal components using the whitening matrix
            X1 = np.dot(whitening, X)
            X1 *= np.sqrt(p)

            # Return whitened components, whitening matrix, and mean of components
            return X1, whitening, X_mean
```

3.3 Symmetric Decorrelation

The outputs must be decorrelated to prevent the vectors from converging to the same maxima. Decorrelation occurs after every iteration. There are several methods of decorrelation including deflation, where the indepedent components are estimated one-by-one, and symmetric where all the components are estimated at once. This means that no vectors are privileged over any others.

Symmetric decorrelation is expressed in the following equation:

$$W = (WW^T)^{-\frac{1}{2}}W$$

Where W is the matrix of the vecots and the inverse square root is found from the eigenvalue decomposition. There are also iterative algorithms to determine the symmetric decorrelation.

```
In [5]: # Symmetric decorrelation of un_mixing matrix
        # Ensures no vectors are privileged over others
        def symmetric_decorrelation(un_mixing):

            # Find eigenvalues and eigenvectors of unmixing matrix
            eig_values, eig_vectors = linalg.eigh(np.dot(un_mixing, un_mixing.T))

            # Apply symmetric decorrelation equation
            sym_un_mixing = np.dot(np.dot(eig_vectors * (1 / np.sqrt(eig_values)), eig_v

            return sym_un_mixing
```

4 Parallel ICA

The parallel ica algorithm estimates all of the independent components at once on each iteration. This requires using the symmetric decorrelation on each iteration After estimating the components on one iteration, the Negentropy is calculated. If the Negentropy increase is below the tolerance, the algorithm stops.

```
In [6]: def parallel_ica(X, init_un_mixing, alpha = 1.0, max_iter = 1000, tol = 1e-4, re

            # Symmetric decorrelation of initial un-mixing components
            un_mixing = symmetric_decorrelation(init_un_mixing)


            p = float(X.shape[1])

            # Iteratively update the un-mixing matrix
            for i in range(max_iter):
```

```
            # Function (g) and gradient from the Negentropy approximation
            gwtx, g_wtx = logcosh(np.dot(un_mixing, X), alpha)

            # Decorrelate the un-mixing matrix on every loop
            new_un_mixing = symmetric_decorrelation(np.dot(gwtx, X.T) / p - g_wtx[:,

            # Calculate convergence measure
            lim = max(abs(abs(np.diag(np.dot(new_un_mixing, un_mixing.T))) - 1))

            # Update un-mixing
            un_mixing = new_un_mixing

            # Check for convergence
            if lim < tol:
                break

        else:
            warnings.warn('FastICA algorithm did not converge. Considering increasin
                          'tolerance or increasing the maximum number of iterations.

        # Return the un-mixing matrix
        if return_iter:
            return un_mixing, i + 1
        else:
            return un_mixing
```

5 Fast ICA

There are several desirable properties of the Fast ICA algorithm compared to the traditional methods for ICA.

1. The convergence is cubic (or at the least quadratic) under the ICA data model assumptions. In contrast, stochastic gradient descent methods only converge linearly.
2. There is no step-size parameter to select as in gradient-based methods. This increases usability of the method.
3. The Fast ICA algorithm direclty finds independent components of nearly any non-Gaussian distribution.
4. The method can be optimized using the right non-linearity, g.
5. Implemented is parallel and can be distributed.

First, the input data must be centered and whitened. Then, the initial un-mixing components are randomly intialized. The algorithm then updates the un-mixing components until covergence. Finally, the mixing matrix and sources can be estimated from the un-mixing matrix and the whitening matrix.

```
In [7]: # X = mixing * sources
        # sources = un-mixing * whitening * X
        def perform_fastica(X, n_components, alpha = 1.0, max_iter = 200, tol = 1e-4):

            # Center and Whiten components
            X1, whitening, X_mean = whiten_components(X.T, n_components)

            # initial un_mixing components
            init_un_mixing = np.asarray(np.random.normal(size = (n_components, n_compone

            # Solve ica using the parallel ica algorithm
            un_mixing = parallel_ica(X1, init_un_mixing, alpha, max_iter, tol)

            # Calculate the sources
            sources = np.dot(np.dot(un_mixing, whitening), X.T).T

            # Calculate the mixing matrix
            w = np.dot(un_mixing, whitening)
            mixing = linalg.pinv(w)

            # Return mixing matrix, sources, and mean of X
            return mixing, sources, X_mean
```

5.1 Inverse ICA Transform

The inverse transform can be used to re-construct the original matrix from the estimated sources and mixing matrix. The mean must also be added back in to find the original features.

```
In [8]: def inverse_fastica(mixing, source, X_mean):
            # Inverse transform
            X = np.dot(sources, mixing.T)
            # Add back in mean
            X += X_mean

            return X
```

6 Conclusions

In this notebook we implemented Independent Component Analysis, a method of blind source separation that finds the independent components assuming that a signal is a linear combination of non-Gaussian sources. We used the FastICA implementation of independent component analysis because of its advantages relative to gradient-based methods.