Verifiable Principal Components Analysis

?

Verifiable Principal Components Analysis

The Principal Component Analysis (PCA) method is an unsupervised learning algorithm that aims to reduce the dimensionality of a dataset consisting of a large number of interrelated variables, while at the same time preserving as much of the variation present in the original dataset as possible. This is achieved by transforming to a new set of variables, the principal components (PC), which are uncorrelated and are ordered in such a way that the first ones retain most of the variation present in all the original variables. More formally, with PCA, given! n n observations of! p p variables, it seeks the possibility of adequately representing this information with a smaller number of variables, constructed as linear combinations of the original variables.

Below, we provide a brief review of the implementation of a Principal Component Analysis (PCA) in Python, which we will then convert to Cairo to transform it into a verifiable ZKML (Principal Component Analysis), using the Orion library. This provides an opportunity to become familiar with the main functions and operators that the framework offers for the implementation of PCA.

Content overview:

Used DataSet

For the purposes of this tutorial, we will use the iris dataset from sklearn.datasets.

```

Copy importnumpyasnp importmath importmatplotlib.pyplotasplt importpandasaspd fromsklearn.datasetsimportload_iris

data=load_iris() X=data['data']

y=data['target']

```

For the purposes of this tutorial, we will not take into account the total number of records in the original dataset. In this sense, we will only focus on the first 105 individuals and the first 3 variables, in order to have comparable results between the python and cairo implementations, taking into consideration the same number of iterations in both programs to achieve orthogonality between the components at the computational level. Therefore, we will have the total number of individuals of the species versicolor and virginica, partially the individuals of the species setosa and with the exclusion of the variable petal width.

```

Copy X,y=X[:105,0:3],y[:105]

```

Before applying PCA, it is important to standardize the data. This ensures that each feature has an equal weight in the calculation of principal components.

```

Copy sigma=np.std(X, axis=0) mu=np.mean(X, axis=0) X_std=(X-mu)/sigma

```

Implementation of the Jacobi algorithm

The Jacobi algorithm is an iterative method for finding the eigenvalues and eigenvectors of a symmetric matrix, which in our case is the correlation matrix over! X s t d \mathbf{X_{std}} . With this method, the aim is to identify pairs of elements off the main diagonal of the matrix and "rotate" them to zero using orthogonal transformations. The idea is that, after enough rotations, the matrix will converge to a diagonal matrix whose diagonal elements will be the eigenvalues of the original matrix. The eigenvectors, on the other hand, are constructed from the rotation matrices applied during the process.

```

Copy defextract_diagonal(matrix): return[matrix[i][i]foriinrange(len(matrix))]
```

```

Copy deffind_max_off_diag(A): n=A.shape[0] p,q=0,1 max_val=abs(A[p, q]) foriinrange(n-1): forjinrange(i+1, n): ifabs(A[i, j])>max_val: max_val=abs(A[i, j]) p,q=i,j returnp,q

defjacobi_eigensystem(A,tol=1e-2,max_iter=500): iflen(A.shape)!=2orA.shape[0]!=A.shape[1]: raiseValueError("A must be a square matrix")

n=A.shape[0] V=np.eye(n)

for_inrange(max_iter): p,q=find_max_off_diag(A)

ifabs(A[p, q])<tol: break

ifA[p,p]==A[q,q]: theta=math.pi/4 else: theta=0.5math.atan(2A[p, q]/(A[p, p]-A[q, q]))

J=np.eye(n) J[p,p],J[q,q]=math.cos(theta),math.cos(theta) J[p,q],J[q,p]=math.sin(theta),-math.sin(theta)

A=np.matmul(np.matmul(J.T,A),J) V=np.matmul(V,J)

returnextract_diagonal(A),V
```

Correlation Matrix

At this point, we determine the correlation matrix, based on the calculations of the variance and covariance matrix for! X s t d \mathbf{X_{std}} .

We compute the covariance matrix

! S = X T X / n − 1 \mathbf{S} = \mathbf{X}^{T} \mathbf{X} /\mathbf{n-1}

Then, we determine the correlation matrix :

! r = C o v (X , Y) / S X S Y \mathbf{r} = \mathbf{Cov(X,Y)} /\mathbf{S_{X} S_{Y}} ,

where! S X S Y \mathbf{S_{X} S_{Y}} are the standard deviations of X and Y respectively.

```

Copy n=(X_std.shape[0]-1) cov_matrix=np.dot(X_std.T,X_std)/n stddevs=np.sqrt(extract_diagonal(cov_matrix))

    array([[1.00961538,-0.20656485,0.83469099], [-0.20656485,1.00961538,-0.57369635], [0.83469099,-0.57369635,1.00961538]])
```

```

Copy corr_matrix=cov_matrix/(stddevs.reshape(-1,1)@stddevs.reshape(1,-1))

corr_matrix

    array([[1. ,-0.20459756,0.82674155], [-0.20459756,1. ,-0.56823258], [0.82674155,-0.56823258,1. ]])
```

```

Copy evalu,evec=jacobi_eigensystem(corr_matrix)
```

```

At this point, we have the eigenvalues and eigenvectors associated with the correlation matrix. Now, we sort the eigenvalues in decreasing order, as the largest of these will be associated with the component that explains the most variability in the data. Consequently, the principal components will be sorted in the same order as the eigenvalues and eigenvectors.

```

Copy idx=np.argsort(evalu)[::-1] evec=evec[:,idx] evalu=evalu[idx]

```

```

Copy evalu,evec

    (array([2.10545934,0.81005256,0.0844881]), array([[-0.58699831,0.55819468,-0.58638867], [0.45714577,0.82631669,0.32896575], [-0.66816968,0.07496276,0.74022284]]))

```

Loadings PCA

Once the aforementioned order is established, we find the loadings which are represented by the discovered eigenvectors (evec). These loadings represent the coefficients of each variable in each of the principal components.

```

Copy loadings=pd.DataFrame(evec,columns=['PC1','PC2','PC3'], index=data['feature_names'][:3]) loadings

    PC1 PC2 PC3 sepallength(cm)-0.5869980.558195-0.586389 sepalwidth(cm)0.4571460.8263170.328966 petallength(cm)-0.6681700.0749630.740223

```

New axis, Principal Components

Next, we identify the new axes or principal components, which are obtained as a linear combination of the standardized original variables.

! P C i = a i 1 X 1 + a i 2 X 2 + . . . + a i n X n \mathbf{PC_{i}} = \mathbf{a_{i1}X_1} + \mathbf{a_{i2}X_2} + \mathbf{...} + \mathbf{a_{in}X_n} ,

Where,

! X 1 , X 2 , . . . + X n \mathbf{X_1}, \mathbf{X_2}, \mathbf{...} + \mathbf{X_n} are the standardized original variables.

! a i 1 , a i 2 , . . . + a i n \mathbf{a_{i1}}, \mathbf{a_{i2}}, \mathbf{...} + \mathbf{a_{in}} are the coefficients or loadings of the! i – t h \mathbf{i-th} eigenvector.

```

Copy principalDf=pd.DataFrame(np.dot(X_std,loadings)) principalDf.columns=["principal component{}".format(i+1)foriinrange(principalDf.shape[1])] principalDf

    principal component1principal component2principal component3 01.4627740.289653-0.115854 11.151659-0.763285-0.290054 21.568707-0.583768-0.019975 31.472596-0.8353030.095131 41.6494870.3818580.043347 ... ... 100-1.8064121.1759440.900363 101-1.555969-0.3514780.487362 102-2.7675431.317228-0.069714 103-2.0230980.4493130.425579 104-2.1903910.8049820.414940 105rows ×3columns

```

Next, we validate the orthogonality between the principal components, as we observe the lack of correlation between these new variables (Principal Components).

```

Copy n=principalDf.shape[0]-1 cov_new=np.dot(np.array(principalDf).T,np.array(principalDf))/n stddevs=np.sqrt(extract_diagonal(cov_new)) corr_new=cov_new/np.matmul(stddevs.reshape(-1,1),stddevs.reshape(1,-1))

```
```

```

Copy corr_new

                array([[1. ,-0.002632,-0.01975951], [-0.002632,1. ,0.0200618], [-0.01975951,0.0200618,1. ]])

```

```

Copy new_corr=round(abs(pd.DataFrame(corr_new))) new_corr

                012 01.00.00.0 10.01.00.0 20.00.01.0

```

The implementation of Jacobi in Python is carried out considering 500 iterations, in order to optimize its implementation at the Cairo level. That is why rounding is applied when checking for orthogonality.

Selection of the number of components to be retained

Each principal component accounts for a proportion of the total variance, and such proportion can be determined by the ratio of each eigenvalue to the total sum of all eigenvalues. Thus, the percentage of variance explained by the i-th component is given by:

! λ i ∑ j = 1 p λ j \frac{\lambda_i}{\sum_{j=1}^{p} \lambda_j}

```

Copy plt.plot(np.cumsum(evalu)/np.sum(evalu)) plt.xlabel('number of components') plt.ylabel('cumulative explained variance');

```

?

```

Copy select_pc=round(((evalu)/np.sum(evalu))[:2].sum(),2)*100 select_pc

                97.0

```

As observed in the previous graph, we decided to keep thefirst 2 components , which explain97% of the total variability of the data.

```

Copy fig=plt.figure(figsize=(8,8)) ax=fig.add_subplot(1,1,1) ax.set_xlabel('Principal Component 1', fontsize=15) ax.set_ylabel('Principal Component 2', fontsize=15) ax.set_title('First two Components of PCA', fontsize=20)

targets=[0,1,2] names=['setosa','versicolor','virginica'] colors=['r','g','b'] fortarget,color,nameinzip(targets, colors, names): indicesToKeep=y==target ax.scatter(principalDf.loc[indicesToKeep,'principal component 1'] , principalDf.loc[indicesToKeep,'principal component 2'] , c=color , s=50) ax.legend(names) ax.grid()

```

?

Based on what is observed in the graph of the first 2 principal components, we notice how the setosa species differentiates from the versicolor and virginica species in principal component 1, which is attributed to the variables petal length (cm), and sepal length (cm).

Among its other applications, here we were able to use PCA to describe a dataset in a dimension smaller than that of the original dataset. As previously discussed, we noticed how we can describe interesting aspects of the original data without the need to address separately all the dimensions of such data.

Convert your model to Cairo

Now that we have a good understanding of the PCA algorithm and their key functions, we will replicate the entire algorithm in Cairo to make it fully verifiable. Since we will be rebuilding the algorithm from scratch, this will be a good opportunity to get acquainted with Orion's built-in functions and the operators that make the transition to Cairo seamless.

Create a new Scarb project

Scarb is the Cairo package manager specifically created to streamline our Cairo and Starknet development process. Scarb will typically manage project dependencies, the compilation process (both pure Cairo and Starknet contracts), downloading and building external libraries to accelerate our development with Orion.You can find all information about Scarb and Cairo installationhere .

To create a new Scarb project, open your terminal and run:

```

Copy scarb new verifiable_principal_component_analysis

```

A new project folder will be created for you and make sure to replace the content in Scarb.toml file with the following code:

```

Copy [package] name="scarb new verifiable_principal_components_analysis" version="0.1.0"

[dependencies] orion={ git="https://github.com/gizatechxyz/orion.git",rev="v0.1.7"}

```

Gerating the dataset in Cairo

Now let's generate the necessary files to begin our transition to Cairo. In our Jupyter Notebook, we will run the necessary code to convert our iris dataset obtained from sklearn.datasets into fixed point values and represent our X, and y values as fixed point tensors in Orion. For the purposes of the tutorial, we will work directly with the Xstd data obtained from python, so we will also convert these to fixed point values.

```

Copy importos

```

```

Copy os.makedirs("src/generated", exist_ok=True)

```

```

Copy tensor_name=["X","X_std","y"]

defgenerate_cairo_files(data,name):

withopen(os.path.join('src','generated',f"{name}.cairo"),"w")asf: f.write( "use array::{ArrayTrait, SpanTrait};\n"+ "use orion::operators::tensor::{core::{Tensor, TensorTrait}};\n"+ "use orion::operators::tensor::FP16x16Tensor;\n"+ "use orion::numbers::fixed_point::implementations::fp16x16::core::{FP16x16, FixedTrait};\n"+ "\n"+f"fn{name}() -> Tensor"+" {\n\n"+ "let mut shape = ArrayTrait::new();\n" ) fordimindata.shape: f.write(f"shape.append({dim});\n")

f.write("let mut data = ArrayTrait::new();\n") forvalinnp.nditer(data.flatten()): f.write(f"data.append(FixedTrait::new({abs(int(decimal_to_fp16x16(val)))},{str(val<0).lower()}));\n") f.write( "let tensor = TensorTrait::::new(shape.span(), data.span());\n"+ "return tensor;\n}" )

withopen(f"src/generated.cairo","w")asf: fornintensor_name: f.write(f"mod{n};\n")

generate_cairo_files(X,"X") generate_cairo_files(X_std,"X_std") generate_cairo_files(y,"y")

```

The X, X_std and y tensor values will now be generated undersrc/generated directory.

Insrc/lib.cairo replace the content with the following code:

```

Copy modgenerated; modhelper; modtest;
```

```

This will tell our compiler to include the separate modules listed above during the compilation of our code. We will be covering each module in detail in the following section, but let's first review the generated folder files.

```

Copy usearray::{ArrayTrait,SpanTrait}; useorion::operators::tensor::{core::{Tensor,TensorTrait}}; useorion::operators::tensor::FP16x16Tensor; useorion::numbers::fixed_point::implementations::fp16x16::core::{FP16x16,FixedTrait};

fnX_std()->Tensor { letmutshape=ArrayTrait::new(); shape.append(105); shape.append(3); letmutdata=ArrayTrait::new();

// data has been truncated (only showing the first 5 values out of the 100 values) modif datos

data.append(FixedTrait::new(41223,true)); data.append(FixedTrait::new(57011,false)); data.append(FixedTrait::new(68250,true)); data.append(FixedTrait::new(61079,true)); data.append(FixedTrait::new(13084,true)); lettensor=TensorTrait::::new(shape.span(), data.span()); returntensor; }

```

Since Cairo does not come with built-in fixed points we have to explicitly define it for our X, y and Xstd values. Luckily, this is already implemented in Orion for us as a struct as shown below:

```

Copy // Example of a FP16x16. structFP16x16{ mag:u32, sign:bool }

```

For this tutorial, we will use fixed point numbers FP16x16 where the magnitude represents the absolute value and the boolean indicates whether the number is negative or positive. In a 16x16 fixed-point format, there are 16 bits dedicated to the integer part of the number and 16 bits for the fractional part of the number. This format allows us to work with a wide range of values and a high degree of precision for conducting the Tensor operations. To replicate the key functions of PCA, we will conduct our operations using FP16x16 Tensors which are also represented as a structs in Orion.

```

Copy structTensor { shape:Span, data:Span }

```

ATensor in Orion takes a shape and a span array of the data.

Implementing PCA using Orion

At this stage, we will be reproducing the Principal component analysis functions now that we have generated our X, y and Xstd Fixedpoint Tensors. We will begin by creating a separate file for our PCA functions file namedhelper.cairo to host all of our core functions.

PCA Core functions

```

Copy usetraits::TryInto; usealexandria_data_structures::array_ext::{SpanTraitExt}; usearray::{ArrayTrait,SpanTrait}; useorion::operators::tensor::{Tensor,TensorTrait}; useorion::numbers::fixed_point::{core::{FixedTrait}};

useorion::operators::tensor::{FP16x16Tensor,FP16x16TensorDiv}; useorion::numbers::fixed_point::implementations::fp16x16::core::{ FP16x16,FP16x16Impl,FP16x16Add,FP16x16AddEq,FP16x16Sub,FP16x16Mul,FP16x16MulEq, FP16x16TryIntoU128,FP16x16PartialEq,FP16x16PartialOrd,FP16x16SubEq,FP16x16Neg,FP16x16Div, FP16x16IntoFelt252,FP16x16Print,HALF };

useorion::numbers::fixed_point::implementations::fp16x16::math::trig;

[derive(Copy,Drop)]

structEigenValues { p_index:usize, q_index:usize, theta:FP16x16, }

fndiv_by_scalar(self:@Tensor, divisor:u32)->Tensor { letmutdata=(*self).data; letmutdata_array=ArrayTrait::new();

```
loop{ matchdata.pop_front() { Option::Some(elem)=>{ data_array.append(FixedTrait::new(elem.mag/divisor,elem.sign)); },
Option::None(_)=>{ breakTensorTrait::::new((*self).shape, data_array.span()); } }; } }

fndiv_by_fp(self:@Tensor, divisor:FP16x16)->Tensor { letmutdata=(*self).data; letmutdata_array=ArrayTrait::new();

loop{ matchdata.pop_front() { Option::Some(elem)=>{ data_array.append(FP16x16Div::div(elem, divisor)); },
Option::None(_)=>{ breakTensorTrait::::new((self).shape, data_array.span()); } }; } }

// find_max_off_diag: Finds the maximum off-diagonal element in a square Tensor. fnfind_max_off_diag(a:@Tensor)->
(usize,usize) { letmutdata=a.data; letmutshape=a.shape;

letn=(a).shape.at(0);

letmuti=0_usize; letmutj=0_usize; letmutp=0_usize; letmutq=1_usize;

letmutmax_val=FixedTrait::abs((*a).at(indices:array![p, q].span()));

loop{ ifi==n { break(p, q); };

j=i+1;

loop{ ifj==n { break; }; ifFixedTrait::abs((a).at(indices:array![i, j].span()))>max_val {
max_val=FixedTrait::abs((a).at(indices:array![i, j].span())); p=i; q=j; }; j+=1; }; i+=1; } }

// jacobi_eigensystem: Implements the Jacobi eigenvalue algorithm to compute the eigenvalues and eigenvectors of a
symmetric Tensor. fnjacobi_eigensystem( muta:Tensor, tol:FP16x16, max_iter:usize )->(Tensor,Tensor) { assert( !
((a).shape.len()!=2_usize||((a).shape.at(0_usize)!=(a).shape.at(1_usize))), 'amust be a square matrix' );

// let two = FixedTrait::new(ONE, false) + FixedTrait::new(ONE, false); let two = FixedTrait::ONE() + FixedTrait::ONE(); let
four = two * two; let half = FixedTrait::::new(HALF, false); let pi = FixedTrait:::new(trig::PI, false);

let mut data = a.data; let mut shape = a.shape; let numRows = *((shape).at(0)); let mut v = eye(numRows: numRows);

let mut i: usize = 0;

loop { let (p, q) = find_max_off_diag(@a);

if i == max_iter || FixedTrait::abs((a).at(indices: array![p, q].span())) < tol { break (extract_diagonal(@a), v); };

let theta = if (a) .at(indices: array![p, p].span()) == (a) .at(indices: array![q, q].span()) { FP16x16Div::div(pi, four) } else { half *
trig::atan( FP16x16Div::div( two * (a).at(indices: array![p, q].span()), (FP16x16Sub::sub( (a).at(indices: array![p, p].span()),
(a).at(indices: array![q, q].span()) )) ) ) };

let eigensystem = EigenValues { p_index: p, q_index: q, theta: theta };

let j_eye = eye(numRows: numRows);

let j = update_eigen_values(self: @j_eye, eigensystem: eigensystem);

let transpose_j = j.transpose(axes: array![1, 0].span()); a = transpose_j.matmul(@a).matmul(@j);

v = v.matmul(@j);

i += 1; } }

// eye: Generates an identity Tensor of the specified size fn eye(numRows: usize) -> Tensor{ let mut data_array =
ArrayTrait::new();

let mut x: usize = 0;

loop { if x == numRows { break; };

let mut y: usize = 0;

loop { if y == numRows { break; };

if x == y { data_array.append(FixedTrait::ONE()); } else { data_array.append(FixedTrait::ZERO()); };

y += 1; }; x += 1; };

Tensor:: { shape: array![numRows, numRows].span(), data: data_array.span() } }

// extract_diagonal: Extracts the diagonal elements from a square tensor fn extract_diagonal(self: @Tensor) -> Tensor { let
```

```
mut data = (*self).data; let mut data_array = ArrayTrait::new();

let dims = (*self).shape.at(0);

let mut x: usize = 0;

loop { if x == *dims { break; };

let mut y: usize = 0;

loop { if y == *dims { break; };

match data.pop_front() { Option::Some(elem) => { if x == y { data_array.append(*elem); }; }, Option::None(_) => { break; } }; y += 1; }; x += 1; };

Tensor:: { shape: array![*dims].span(), data: data_array.span() } }

// update_eigen_values: Updates the eigenvalues of a symmetric tensor fn update_eigen_values( self: @Tensor, eigensystem: EigenValues ) -> Tensor { let mut data = (*self).data; let mut data_array = ArrayTrait::new();

let mut x: usize = 0; let mut y: usize = 0; let mut index: usize = 0; let dims =s(elf).shape.at(0); let items =dims * dims; let dims_y = (self).shape.at(1);

loop { if index == items { break; };

if y == *dims_y { x += 1; y = 0; };

match data.pop_front() { Option::Some(elem) => { let eigen_values = eigensystem;

let value = if (eigen_values.p_index, eigen_values.p_index) == (x, y) { trig::cos(eigen_values.theta) } else if (eigen_values.q_index, eigen_values.q_index) == (x, y) { trig::cos(eigen_values.theta) } else if (eigen_values.p_index, eigen_values.q_index) == (x, y) { trig::sin(eigen_values.theta) } else if (eigen_values.q_index, eigen_values.p_index) == (x, y) { trig::sin(-eigen_values.theta) } else { *elem };

data_array.append(value); y += 1; index += 1; }, Option::None(_) => { break; } }; };

Tensor:: { shape: *self.shape, data: data_array.span() } }

// check_unit_diagonal_tensor: Checks whether a square tensor has a unit diagonal fn check_unit_diagonal_tensor(self: @Tensor) -> bool { let mut x: usize = 0; let mut valid: bool = true; let dim_x =s(elf).shape.at(0); let dim_y = (self).shape.at(1);

loop { if x == *dim_x || !valid { break valid; };

let mut y: usize = 0;

loop { if y == *dim_y { break; };

if x == y { if (self).at(indices: array![x, y].span()) != FixedTrait::ONE() { valid = false; break; } } else { if (self).at(indices: array![x, y].span()) != FixedTrait::ZERO() { valid = false; break; } };

y += 1; }; x += 1; } }
```

For the purposes of this tutorial, we will apply sorting to the tensors evec and evalu obtained from the execution of our Cairo jacobi_eigensystem function using the functions provided by NumPy for this purpose, generating the evalu_sort.cairo and evec_sort.cairo files

```
Copy evalu_from_cairo=np.array([52513,137534,5393]) evec_from_cairo=np.array([ [36422,-38024,-38467], [53777,30012,21440], [5195,-43789,48143] ]) idx=np.argsort(evalu_from_cairo)[::-1] evec_ord=evec_from_cairo[:,idx] evalu_ord=np.sort(evalu_from_cairo)[::-1]

          [137534525135393] [[-3802436422-38467] [300125377721440] [-43789519548143]]
```

Testing the model

Now that we have implemented all the necessary functions for PCA, we can finally test our dimensionality reduction algorithm. We begin by creating a new separate test file namedtest.cairo and import all the necessary Orion libraries, including our X values and y and Xstd values found in the generated folder. We also import all the key functions for PCA from thehelper.cairo file, as we will rely on them to construct the model.

```

Copy

## [cfg(test)]

modtests { usetraits::TryInto; usealexandria_data_structures::array_ext::{SpanTraitExt}; usearray::{ArrayTrait,SpanTrait}; useorion::operators::tensor::{Tensor,TensorTrait}; useorion::numbers::fixed_point::{core::{FixedTrait}};

useorion::operators::tensor::{FP16x16Tensor,FP16x16TensorDiv,FP16x16TensorSub};

useorion::numbers::fixed_point::implementations::fp16x16::core::{ FP16x16,FP16x16Impl,FP16x16Add,FP16x16AddEq,FP16x16Sub,FP16x16Mul, FP16x16MulEq,FP16x16TryIntoU128,FP16x16PartialEq,FP16x16PartialOrd,FP16x16SubEq, FP16x16Neg,FP16x16Div,FP16x16IntoFelt252,FP16x16Print,FP16x16TryIntoU32 };

usepca::{ helper::{ EigenValues, extract_diagonal, eye, find_max_off_diag, jacobi_eigensystem, update_eigen_values, check_unit_diagonal_tensor, div_by_scalar, div_by_fp } };

usepca::{generated::{X_std::X_std, X::X, y::y, evalu_sort::evalu_sort, evec_sort::evec_sort}};

## [test]

## [available_gas(99999999999999999)]

fnpca_test() { lettol=FixedTrait:::new(655,false);// 655 is 0.01 = 1e-2 letmax_iter=500_usize;

letX_std=X_std(); letX=X(); lety=y();

letmutn:usize=*((X_std).shape.at(0))-1; letsize=*(X_std.shape.at(1));

letX_std_transpose=X_std.transpose(axes:array![1,0].span()); letmutcov_matrix=div_by_scalar(@(X_std_transpose.matmul(@X_std)), n);

letmutstddevs=extract_diagonal(@cov_matrix).sqrt();

letmutstddevs_left=stddevs.reshape(array![size,1].span()); letmutstddevs_right=stddevs.reshape(array![1, size].span()); letcorr_matrix=cov_matrix/stddevs_left.matmul(@stddevs_right);

let(evalu, evec)=jacobi_eigensystem(a:corr_matrix, tol:tol, max_iter:max_iter);

let(evalu, evec)=(evalu_sort(),evec_sort());

letloadings=evec;

letprincipal_component=X_std.matmul(@loadings);

n=*((principal_component).shape.at(0))-1;

letprincipal_component_transpose=principal_component .transpose(axes:array![1,0].span());

letcov_new=div_by_scalar( @(principal_component_transpose.matmul(@principal_component)), n );

stddevs=extract_diagonal(@cov_new).sqrt(); stddevs_left=stddevs.reshape(array![size,1].span()); stddevs_right=stddevs.reshape(array![1, size].span()); letcorr_new=cov_new/stddevs_left.matmul(@stddevs_right);

letnew_corr=(@corr_new.abs()).round();

// The orthogonality of the tensor is validated. assert(check_unit_diagonal_tensor(@new_corr), 'orthogonalityis invalid');

let evalu_cumsum = evalu.cumsum(0, Option::None(()), Option::None(())); let sum = evalu_cumsum.data.at(evalu_cumsum.data.len() - 1); let evalu_div_sum = div_by_fp(@evalu, *sum); let pc =* (*evalu_div_sum.data.at(0) + *evalu_div_sum.data.at(1)) * FixedTrait:::new_unscaled(100, false);
```

assert(FixedTrait::round(pc).mag == 0x610000, 'nomatchwith notebook version'); // 0x610000 == 97

} }

```

Our model will be tested using thepca_test() function, which will follow these steps:

1. Data retrieval: The function starts by obtaining the X and y feature values with their labels, both coming from the generated folder.
2. Construction of correlation matrix: Once we have the data, we proceed to calculate our correlation matrix on the standardized X data.
3. Determination of eigenvalues and eigenvectors : After running the jacobi_eigensystem function, we obtain our jacobi_eigensystem eigenvalues and eigenvectors.
4. PC identification phase : In this phase, we express the PCs as a linear combination of the original variables.
5. Orthogonality: Once the PCs are obtained, we validate the orthogonality between them.
6. Determination of the number of PCs to be retained : At this point, we evaluate according to the variability captured by each PC, the number of principal components to be retained.
7.

Finally, we can execute the test file by runningscarb test

```

Copy scarbtest testingpca... running1tests testpca::test::tests::pca_test...ok(gasusageest.:3575660610) testresult:ok.1passed;0failed;0ignored;0filteredout;

```

And as we can our test cases have passed!

If you've made it this far, well done! You are now capable of building verifiable ML models, making them ever more reliable and transparent than ever before.

We invite the community to join us in forging a future in making AI transparent and reliable resource for all.

Previous Verifiable Support Vector Machine Next Provable MLR: Forecasting AAVE's Lifetime Repayments

Last updated2 months ago
```