chapter-2-1

January 29, 2022

Run in Google Colab

1 Introduction to Machine Learning: Preprocessing

1.1 Definitions

Raw data rarely comes in the form and shape that is necessary for the optimal performance of a learning algorithm. On the other hand, the success of a machine learning algorithm highly depends on the quality of the data fed into the model. Real-world data is often dirty containing outliers, missing values, wrong data types, irrelevant features, or non-standardized data. The presence of any of these will prevent the machine learning model to properly learn. For this reason, transforming raw data into a useful format is an essential stage in the machine learning process. Therefore, it is absolutely critical to ensure that we examine and preprocess a dataset before we feed it to a learning algorithm. In this section, we will discuss the essential data preprocessing techniques that will help us to build good machine learning models.

The topics that we will cover in this lesson are as follows:

- Removing and imputing missing values from the dataset
- Getting categorical data into shape for machine learning algorithms
- Selecting relevant features for the model construction
- Feature Normalization

1.2 Dealing with missing data

The real-world data often has a lot of missing values. The cause of missing values can be data corruption or failure to record data. The handling of missing data is very important during the preprocessing of the dataset as many machine learning algorithms do not support missing values.

Let's create a simple example data frame from a comma-separated values (CSV) file to get a better grasp of the problem:

[2]:
```
import pandas as pd
from io import StringIO

csv_data = \
    '''A,B,C,D
    1.0,2.0,3.0,4.0
    5.0,6.0,,8.0
```

```
    10.0,11.0,12.0,'''
# If you are using Python 2.7, you need
# to convert the string to unicode:
# csv_data = unicode(csv_data)
df = pd.read_csv(StringIO(csv_data))
df
```

[2]:

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	NaN	8.0
2	10.0	11.0	12.0	NaN

1.2.1 Delete Rows with Missing Values

One of the easiest ways to deal with missing data is simply to remove the corresponding features (columns) or training examples (rows) from the dataset entirely; Missing values can be handled by deleting the rows or columns having null values. If columns have more than half of the rows as null then the entire column can be dropped. The rows which are having one or more columns values as null can also be dropped.

Remember that, in pandas, rows with missing values can easily be dropped via the dropna method:

[3]:
```
df.dropna(axis=0)
```

[3]:

	A	B	C	D
0	1.0	2.0	3.0	4.0

Although the removal of missing data seems to be a convenient approach, it also comes with certain disadvantages; for example, we may end up removing too many samples, which will make a reliable analysis impossible. Or, if we remove too many feature columns, we will run the risk of losing valuable information that our classifier needs to discriminate between classes. In the next section, we will look at one of the most commonly used alternatives for dealing with missing values: interpolation techniques.

Pros: - A model trained with the removal of all missing values creates a robust model.

Cons: - Loss of a lot of information. - Works poorly if the percentage of missing values is excessive in comparison to the complete dataset.

1.2.2 Imputing missing values

One of the most common interpolation techniques is mean imputation, where we simply replace the missing value with the mean value of the entire feature column. A convenient way to achieve this is by using the **SimpleImputer** class from scikit-learn, as shown in the following code:

[4]:
```
from sklearn.impute import SimpleImputer
import numpy as np

imr = SimpleImputer(missing_values=np.nan, strategy='mean')
```

```
imr = imr.fit(df.values)
imputed_data = imr.transform(df.values)
imputed_data
```

[4]:
```
array([[ 1. ,  2. ,  3. ,  4. ],
       [ 5. ,  6. ,  7.5,  8. ],
       [10. , 11. , 12. ,  6. ]])
```

Alternatively, an even more convenient way to impute missing values is by using pandas' **fillna** method and providing an imputation method as an argument. For example, using pandas, we could achieve the same mean imputation directly in the DataFrame object via the following command:

[5]:
```
df.fillna(df.mean())
```

[5]:

	A	B	C	D
0	1.0	2.0	3.0	4.0
1	5.0	6.0	7.5	8.0
2	10.0	11.0	12.0	6.0

Pros: - Prevent data loss which results in deletion of rows or columns - Works well with a small dataset and is easy to implement.

Cons: - Works only with numerical continuous variables. - Can cause data leakage - Do not factor the covariance between features.

1.3 Categorical Data

Categorical data is a form of data that takes on values within a finite set of discrete classes. It is difficult to count or measure categorical data using numbers and therefore they are divided into categories.

When we are talking about categorical data, we have to further distinguish between **ordinal** and **nominal** features.

Ordinal features can be understood as categorical values that *can be sorted or ordered*. For example, t-shirt size would be an ordinal feature, because we can define an order: XL > L > M.

In contrast, **nominal** features don't imply any order and, to continue with the previous example, we could think of t-shirt color as a nominal feature since it typically doesn't make sense to say that, for example, red is larger than blue.

Before we explore different techniques for handling such categorical data, let's create a new DataFrame to illustrate the problem:

[34]:
```
import pandas as pd

df = pd.DataFrame([
    ['green', 'M', 10.1, 'class2'],
    ['red', 'L', 13.5, 'class1'],
    ['blue', 'XL', 15.3, 'class2']])
```

```
df.columns = ['color', 'size', 'price', 'classlabel']
df
```

[34]:

	color	size	price	classlabel
0	green	M	10.1	class2
1	red	L	13.5	class1
2	blue	XL	15.3	class2

To make sure that the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers. Unfortunately, there is no convenient function that can automatically derive the correct order of the labels of our size feature, so we have to define the mapping manually. In the following simple example, let's assume that we know the numerical difference between features, for example, XL = L + 1 = M + 2:

[35]:
```
size_mapping = {'XL': 3,'L': 2,'M': 1}

df2 = df.copy()
df2['size'] = df2['size'].map(size_mapping)
df2
```

[35]:

	color	size	price	classlabel
0	green	1	10.1	class2
1	red	2	13.5	class1
2	blue	3	15.3	class2

[38]:
```
df3 = df.copy()
```

[41]:
```
from sklearn.preprocessing import OrdinalEncoder
ordinal = OrdinalEncoder(categories=[['M', 'L', 'XL']])
df3['size'] = ordinal.fit_transform(df3[['size']])
df3
```

[41]:

	color	size	price	classlabel
0	green	0.0	10.1	class2
1	red	1.0	13.5	class1
2	blue	2.0	15.3	class2

1.3.1 Encoding Class Labels

Many machine learning libraries require that class labels are encoded as integer values. Although most estimators for classification in scikit-learn convert class labels to integers internally, it is considered good practice to provide class labels as integer arrays to avoid technical glitches. To encode the class labels, we can use an approach similar to the mapping of ordinal features discussed previously. We need to remember that class labels are not ordinal, and it doesn't matter which integer number we assign to a particular string label. Thus, we can simply enumerate the class labels, starting at 0:

[8]:
```
import numpy as np

class_mapping = {label: idx for idx, label in enumerate(np.
 ↪unique(df['classlabel']))}
class_mapping
```

[8]:
```
{'class1': 0, 'class2': 1}
```

Next, we can use the mapping dictionary to transform the class labels into integers:

[9]:
```
df['classlabel'] = df['classlabel'].map(class_mapping)
df
```

[9]:

	color	size	price	classlabel
0	green	1	10.1	1
1	red	2	13.5	0
2	blue	3	15.3	1

We can reverse the key-value pairs in the mapping dictionary as follows to map the converted class labels back to the original string representation:

[10]:
```
inv_class_mapping = {v: k for k, v in class_mapping.items()}
df['classlabel'] = df['classlabel'].map(inv_class_mapping)
df
```

[10]:

	color	size	price	classlabel
0	green	1	10.1	class2
1	red	2	13.5	class1
2	blue	3	15.3	class2

Alternatively, there is a convenient LabelEncoder class directly implemented in scikit-learn to achieve this:

[11]:
```
from sklearn.preprocessing import LabelEncoder

class_le = LabelEncoder()
y = class_le.fit_transform(df['classlabel'].values)
y
```

[11]:
```
array([1, 0, 1])
```

When there is no a natural order we have to resort to a different approach that is to use the technique called **one-hot encoding**. The idea behind this approach is to create a new dummy feature for each unique value in the nominal feature column. Here, we would convert the color feature into three new features: *blue*, *green*, and *red*. Binary values can then be used to indicate the particular color of an example; for example, a blue example can be encoded as *blue=1*, *green=0*, *red=0*. To perform this transformation, we can use the OneHotEncoder that is implemented in scikit-learn's preprocessing module:

[12]:
```
from sklearn.preprocessing import OneHotEncoder

X = df[['color', 'size', 'price']].values
color_ohe = OneHotEncoder()
color_ohe.fit_transform(X[:, 0].reshape(-1, 1)).toarray()
```

[12]:
```
array([[0., 1., 0.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

An even more convenient way to create those dummy features via one-hot encoding is to use the get_dummies method implemented in pandas. Applied to a DataFrame, the get_dummies method will only convert string columns and leave all other columns unchanged:

[13]:
```
pd.get_dummies(df[['price', 'color', 'size']])
```

[13]:

	price	size	color_blue	color_green	color_red
0	10.1	1	0	1	0
1	13.5	2	0	0	1
2	15.3	3	1	0	0

Inserire rimando ad esempio Ch. 4-0

1.4 Feature Normalization

Many machine learning algorithms require that the selected features are on the same scale for optimal performance, this process is called "Feature Normalization" and is the subject of this paragraph.

Data Normalization is a common practice in machine learning which consists of transforming numeric columns to a common scale. In machine learning, some feature values differ from others multiple times. The features with higher values will dominate the leaning process. However, it does not mean those variables are more important to predict the outcome of the model. Data normalization transforms multiscaled data to the same scale. After normalization, all variables have a similar influence on the model, improving the stability and performance of the learning algorithm.

There are multiple normalization techniques in statistics. In this notebook, we will cover the most important ones:

- The maximum absolute scaling
- The min-max feature scaling
- The z-score method

1.4.1 The maximum absolute scaling

The maximum absolute scaling rescales each feature between -1 and 1 by dividing every observation by its maximum absolute value.

$$x_{new} = \frac{x_{old}}{\max |x_{old}|}$$

1.4.2 The min-max feature scaling

The min-max approach (often called normalization) rescales the feature to a fixed range of [0,1] by subtracting the minimum value of the feature and then dividing by the range:

$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$

The min-max scaling procedure is implemented in scikit-learn and can be used as follows:

[18]:
```
#
# Here we have to load the file 'salary_vs_age_1.csv'
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[19]:
```
# Load the Pandas libraries with alias 'pd'
import pandas as pd
# Read data from file 'salary_vs_age_1.csv'
# (in the same directory that your python process is based)
# Control delimiters, with read_table
df1 = pd.read_table(path + "salary_vs_age_1.csv", sep=";")
# Preview the first 5 lines of the loaded data
print(df1.head())

columns_titles = ["Salary","Age"]
df2=df1.reindex(columns=columns_titles)
df2

df2['Salary'] = df2['Salary']/1000
df2['Age2']=df2['Age']**2
df2['Age3']=df2['Age']**3
df2['Age4']=df2['Age']**4
df2['Age5']=df2['Age']**5
df2
```

```
   Age  Salary
0   25  135000
1   27  105000
2   30  105000
3   35  220000
4   40  300000
```

[19]:

	Salary	Age	Age2	Age3	Age4	Age5
0	135.0	25	625	15625	390625	9765625

```
1   105.0   27   729   19683    531441    14348907
2   105.0   30   900   27000    810000    24300000
3   220.0   35  1225   42875   1500625    52521875
4   300.0   40  1600   64000   2560000   102400000
5   270.0   45  2025   91125   4100625   184528125
6   265.0   50  2500  125000   6250000   312500000
7   260.0   55  3025  166375   9150625   503284375
8   240.0   60  3600  216000  12960000   777600000
9   265.0   65  4225  274625  17850625  1160290625
```

[20]:
```
from sklearn.preprocessing import MinMaxScaler

mms = MinMaxScaler()
df3 = pd.DataFrame(mms.fit_transform(df2))
df3
```

[20]:

	0	1	2	3	4	5
0	0.153846	0.000	0.000000	0.000000	0.000000	0.000000
1	0.000000	0.050	0.028889	0.015668	0.008065	0.003984
2	0.000000	0.125	0.076389	0.043919	0.024019	0.012633
3	0.589744	0.250	0.166667	0.105212	0.063574	0.037162
4	1.000000	0.375	0.270833	0.186776	0.124248	0.080515
5	0.846154	0.500	0.388889	0.291506	0.212486	0.151898
6	0.820513	0.625	0.520833	0.422297	0.335588	0.263127
7	0.794872	0.750	0.666667	0.582046	0.501718	0.428951
8	0.692308	0.875	0.826389	0.773649	0.719895	0.667377
9	0.820513	1.000	1.000000	1.000000	1.000000	1.000000

1.4.3 Z-Score

The **z-score** method (often called **standardization**) transforms the data into a distribution with a mean of 0 and a standard deviation of 1. Each standardized value is computed by subtracting the mean of the corresponding feature and then dividing by the standard deviation.

$$x_{new} = \frac{x_{old} - \mu}{\sigma}$$

Unlike min-max scaling, the z-score does not rescale the feature to a fixed range. The z-score typically ranges from -3.00 to 3.00 (more than 99% of the data) if the input is normally distributed.

It is important to bear in mind that z-scores are not necessarily normally distributed. They just scale the data and follow the same distribution as the original input. This transformed distribution has a mean of 0 and a standard deviation of 1 and is going to be the standard normal distribution only if the input feature follows a normal distribution.

Standardization can easily be achieved by using the built-in NumPy methods mean and std:

[21]:
```
import numpy as np
```

```
X = np.array([6, 7, 7, 12, 13, 13, 15, 16, 19, 22])

X_std = np.copy(X)
X_std = (X - X.mean()) / X.std()

print(X_std)
```

```
[-1.39443338 -1.19522861 -1.19522861 -0.19920477  0.          0.
  0.39840954  0.5976143   1.19522861  1.79284291]
```

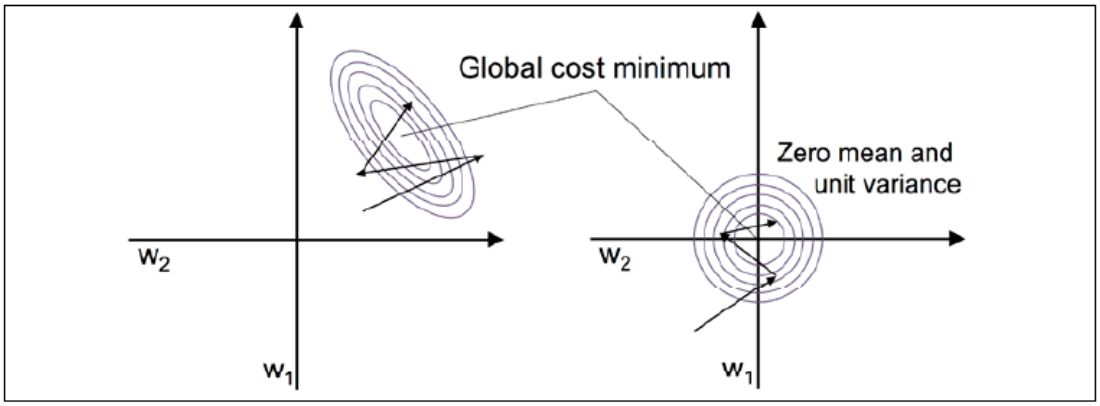
Or simply using the specific function of the stats module of scipy

[22]:
```
import scipy.stats as stats

stats.zscore(X)
```

[22]:
```
array([-1.39443338, -1.19522861, -1.19522861, -0.19920477,  0.        ,
        0.        ,  0.39840954,  0.5976143 ,  1.19522861,  1.79284291])
```

Standardization is very useful with gradient descent learning. In this case the optimizer has to go through fewer steps to find a good or optimal solution (the global cost minimum), as illustrated in the following figure, where the subfigures represent the cost surface as a function of two model weights in a two-dimensional classification problem:



Similar to the MinMaxScaler class, scikit-learn also implements a class for standardization:

[23]:
```
from sklearn.preprocessing import StandardScaler

stdsc = StandardScaler()
df4 = pd.DataFrame(stdsc.fit_transform(df2))
df4
```

[23]:

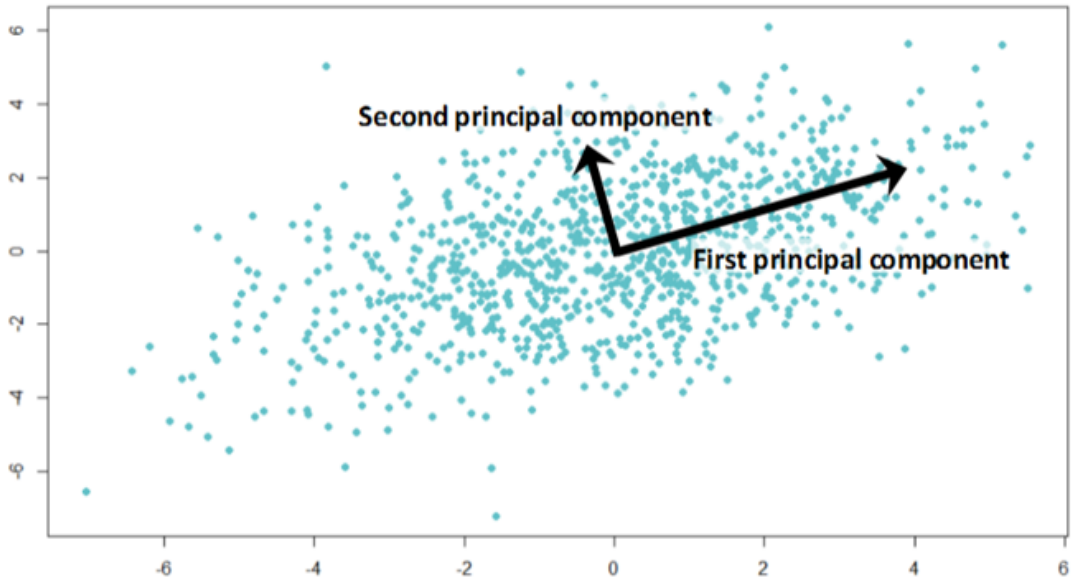
	0	1	2	3	4	5
0	-1.170242	-1.359724	-1.189131	-1.041783	-0.920815	-0.824435
1	-1.601005	-1.210304	-1.102065	-0.994071	-0.895974	-0.812022

```
2 -1.601005 -0.986174 -0.958907 -0.908042 -0.846835 -0.785069
3  0.050256 -0.612623 -0.686823 -0.721391 -0.725003 -0.708630
4  1.198959 -0.239072 -0.372880 -0.473014 -0.538122 -0.573535
5  0.768195  0.134478 -0.017078 -0.154092 -0.266345 -0.351091
6  0.696401  0.508029  0.380582  0.244194  0.112820 -0.004480
7  0.624608  0.881579  0.820102  0.730661  0.624511  0.512260
8  0.337432  1.255130  1.301481  1.314127  1.296512  1.255243
9  0.696401  1.628681  1.824719  2.003411  2.159252  2.291760
```

1.5 Dimensionality Reduction

1.5.1 Principal Component Analysis (PCA)

PCA as a concept is useful for measuring risk arising from a set of correlated market variables. For example, interest rates that are quoted in the market have correlation to each other. Thus, an interest rate for tenor 1Y is not independent of interest rate for tenor say 3Y. There is always some degree of correlation between all the tenor points with each other. To achieve this objective, the PCA model computes a set of variables that are called as Principal components (PCs). PCA is a model which involves transforming a set of observations (i.e. interest rate time series in our case) into a set of uncorrelated variables called as the PCs. This transformation behaves in way such that the first PC explains the largest possible variance, and this accounts for majority of the variability in the data. Each succeeding component in turn explains the highest possible variance while at the same time following the condition of orthogonality to each of the preceding PCs.



Resulting PCs computed by the model are uncorrelated to each other, thereby allowing them to be used independently with respect to each other. Individual PCs are calculated using the concept of Eigen values again a concept of linear algebra. PCs represent directions of the data that explain the maximum amount of variance, i.e. the vectors that capture most of the information that is embedded in the data. The relationship between the variance and information is that, the larger

the variance carried by the vector, the larger the dispersion of the data points along it, and larger the dispersion along the vector, the more information it contains.

PCA algorithm performs **dimensionality reduction** on the data set. ***Dimensionality reduction implies, we attempt to capture the essence of a multivariate dataset into fewer number of variables that would explain the required result***.

So subsequent to generation of individual PCs, only those PCs are selected that explain the maximum variation thereby capturing the essence of the analysis. Machine learning algorithms implementing the concept of Principal Component Analysis (PCA) can be used to this end. A PCA algorithm accepts all of the incoming interest data as an input, and it processes the data so that as an output we get a certain set of interest rate tenor points which contribute to around say 97% to 98% of the risk of our interest rate sensitive portfolio. This is technically termed as dimensionality reduction as mentioned earlier. This substantially reduces the load on the system resources, since now, the system will use only those tenor points as have been chosen by the PCA algorithm. This enables freeing up of valuable system resources which now can be used for other productive purposes. PCA can be implemented in Python.

PCA algorithm involves the following steps:

1. **Standardization**: As we have already seen in this notebook, this step involves standardizing the input variables so that they may be used in the PCA analysis. Accuracy of PCA algorithm is a function of the accuracy of inputs. So, in the very first step of the algorithm, we perform a standardization which results in all variables getting transformed to a same scale. This builds the foundation of the PCA analysis.
2. **Covariance matrix**: This step involves computation of a covariance matrix which gives the relationship between the input variables. A covariance matrix is a symmetric matrix. With the diagonal elements giving the correlation, and the off-diagonal elements giving the covariance between variables. Depending on the sign of the covariance, the algorithm understands whether there is a direct or an inverse relation between variables. This step is important with respect to dimensionality reduction, as highly correlated variables may convey redundant information so the algorithm may appropriately handle these during the analysis.
3. **Eigen algebra**: Eigen values and Eigen vectors are calculated from the covariance matrix computed in the step above. Eigenvectors of the covariance matrix are the direction of the axes where there is most variance i.e. most information. These are the PCs. Eigen values are the coefficients attached to the eigen vectors; they explain the amount of variance carried by each of the PCs. By ranking the eigen vectors in the order of their eigen values, we get the PCs in order of their significance. Next, we choose only the top 2 or top 3 PCs. The reason being, that its these top 2 or top 3 PCs that explain most of the variance in the data. Generally, top 3 PCs explain more than 97% of the variance in the data. Speaking about **Interest Rate Risk**, out of the top 3 PCs the first PC is attributed to account for parallel shifts in the rates, second PC is attributed to account for steepening of the curve, third PC is attributed to account for bowing of the interest rate curve. We also compute the standard deviations of the same called as factor loadings.

[1]:
```
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
```

```
from sklearn.preprocessing import scale
```

[23]:
```
#
# Here we have to load the file 'MarketData.csv'
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[24]:
```
data = pd.read_csv(path + 'MarketData.csv', sep=',')
x = pd.DataFrame(data)
x.head()
```

[24]:

	Date	3m	6m	1y	2y	3y	4y	5y	7y	10y
0	1	7.71	7.90	8.16	8.56	8.71	8.75	8.94	9.04	8.76
1	2	7.75	7.93	8.17	8.58	8.72	8.76	8.96	9.05	8.76
2	3	7.68	7.90	8.18	8.54	8.69	8.75	8.92	9.02	8.73
3	4	7.69	7.93	8.22	8.55	8.70	8.76	8.93	9.03	8.75
4	5	7.69	7.96	8.22	8.55	8.70	8.76	8.93	9.03	8.75

[25]:
```
df = x.drop(axis=1,columns=['Date'])
X = df.values
```

[17]:
```
#Normalization of the data
X = scale(X)
```

Factor loadings can be calculated as below:

[19]:
```
pca = PCA(n_components=9)
pca.fit(X)
factor_loading = pca.components_
df_factor_loading = pd.DataFrame(factor_loading)
df_factor_loading
```

[19]:

	0	1	2	3	4	5	6	\
0	-0.326333	-0.332127	-0.335253	-0.336409	-0.336407	-0.336645	-0.336749	
1	0.544483	0.481083	0.254532	0.026220	-0.094576	-0.232499	-0.195293	
2	-0.420097	-0.089586	0.239718	0.367905	0.368985	0.193024	0.162056	
3	-0.483507	0.155548	0.552238	0.194323	-0.150791	-0.285729	-0.310074	
4	-0.077813	0.031178	-0.068074	-0.189461	-0.066528	0.199704	0.509370	
5	0.265687	-0.373126	-0.286407	0.548879	0.267776	0.002223	-0.390429	
6	0.156875	-0.313745	0.089269	0.445102	-0.474288	-0.417560	0.506926	
7	-0.113533	0.227273	-0.269583	-0.058133	0.562689	-0.692176	0.238268	
8	0.269520	-0.579435	0.541370	-0.417573	0.318277	-0.142704	0.020379	

```
          7         8
0 -0.331415 -0.328477
1 -0.439282 -0.335432
2 -0.254609 -0.598480
3 -0.106423  0.432952
4 -0.714587  0.371542
5 -0.316872  0.286948
6  0.089134 -0.081522
7  0.028055  0.078216
8 -0.035621  0.028572
```

Factor loadings explain the relation between the impact of a factor on interest rates at respective tenor points. In PCA we also analyse the amount of dispersion explained by each of the PCs. Now we will see which PC contributes how much amount of variance/dispersion:

[22]:
```
# variance percent of each PC
variance_percent_df = pd.DataFrame(data=pca.explained_variance_)
variance_ratio_df = pd.DataFrame(data=pca.explained_variance_ratio_)
variance_ratio_df = variance_ratio_df * 100
variance_ratio_df
```

[22]:
```
            0
0  96.406078
1   1.931193
2   1.183464
3   0.242575
4   0.125015
5   0.059899
6   0.032387
7   0.013283
8   0.006106
```

From the table alongside, we observe that PC1 explains almost 96% of the total variation, and PC2 explains close to 1.95% of total variation. Therefore, rather than using all PCs in the subsequent calculation, we can only use PC1 and PC2 in further calculation as these two components explain close to 98% of the total variance.

- PC1 corresponds to the roughly the parallel shift in the yield curve.
- PC2 corresponds to roughly a steepening in the yield curve.

This is in-line with the theory of fixed income risk measurement which states that majority of the movement in the price of a bond is explained by the parallel shift in the yield curve and the residual movements in the price is explained by steepening and curvature of the interest rate curve.

1.6 References

Abhyankar Ameya, "*Exploring Risk Analytics using PCA with Python*", Medium, data files for the interest rate example and further details about the python code can be dowloaded from the github repository of the author here