Machine Learning

Machine Learning

Part 1 – Data Preparation (Missing Values and Encoding Categorical Features)

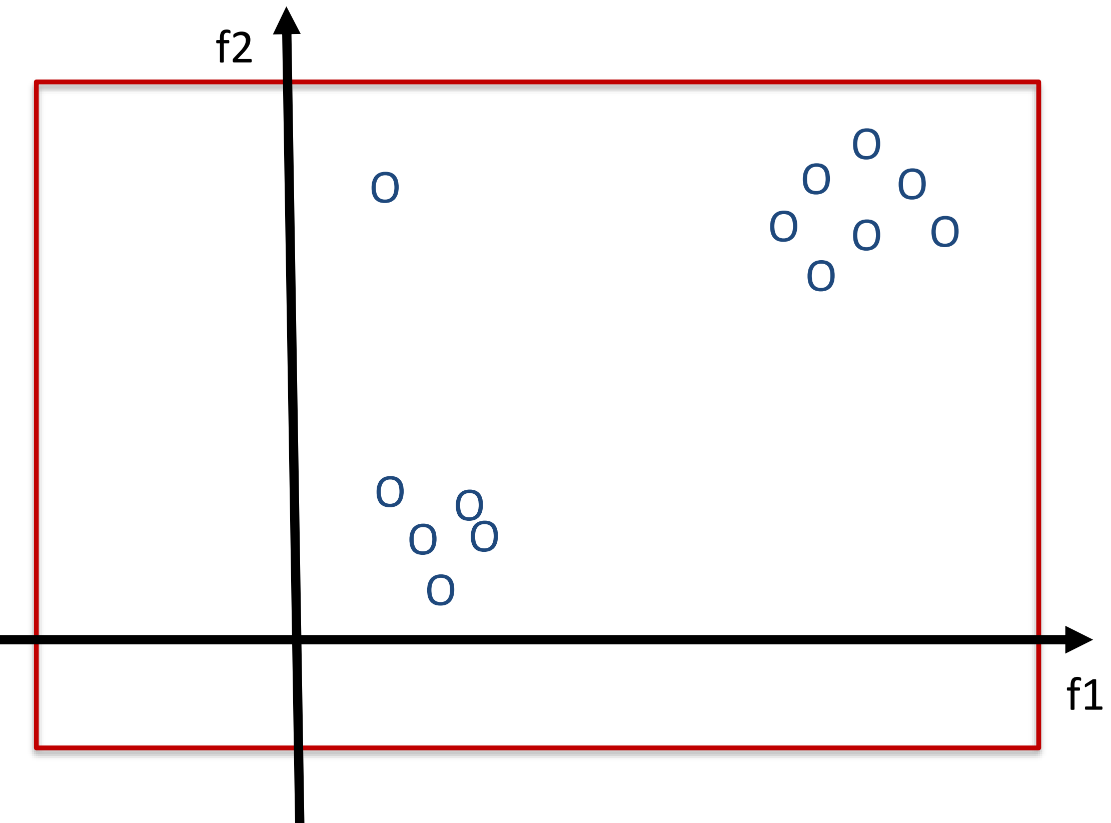
Ted Scully

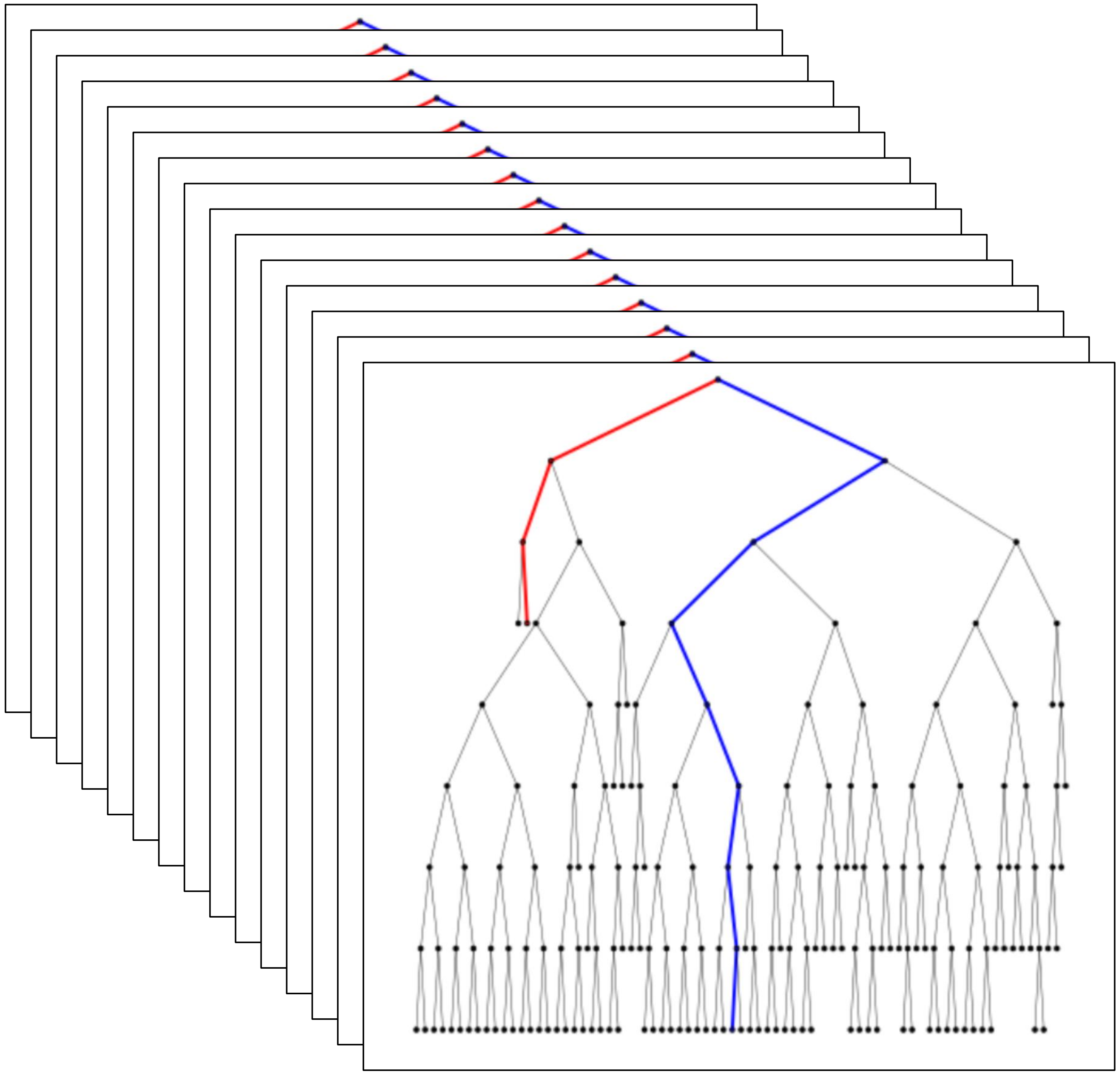
Isolation Forests for Multi-variate Outlier Detection

Now let’s consider a feature space with 2 features. An isolation forest will build many **isolation trees**.

To build each tree we will (1) **randomly select a feature** and will (2) **randomly select a value to partition** that feature. It will continue this process iteratively.

Below f1 is feature 1 and f2 is feature 2 from our dataset. We initially pick f2 at random.



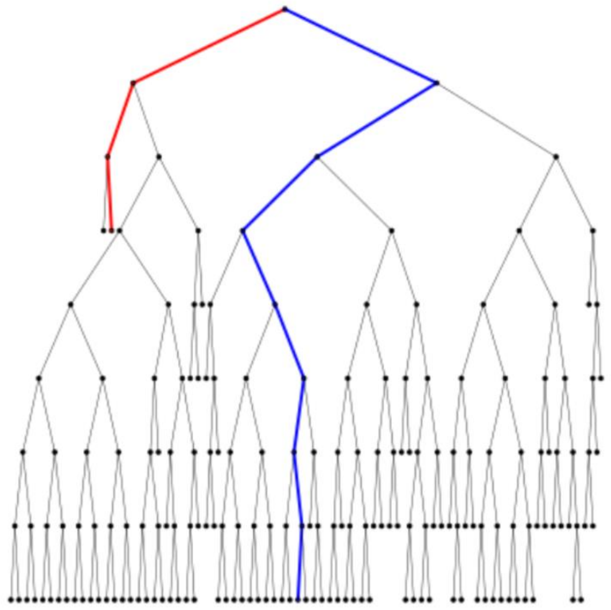


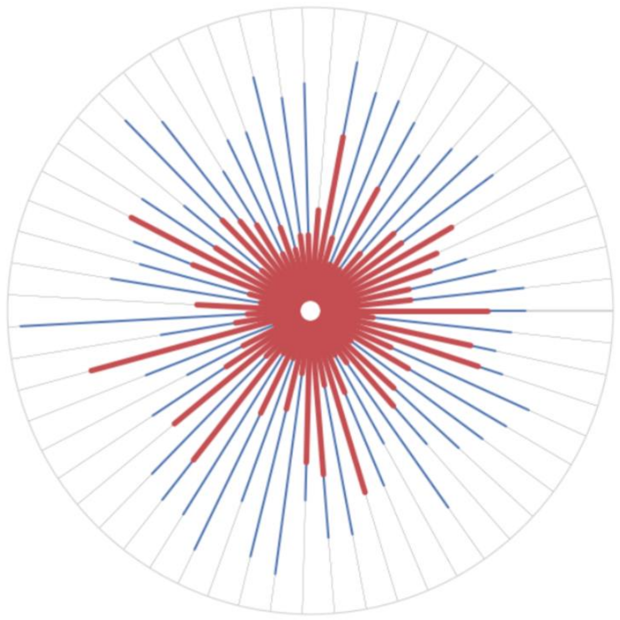
We build many isolation trees.

Each time we go to build a tree a random sample of the original data is extracted and the isolation tree is built using this subset.

The image below (on the right) depicts an isolation forest consisting of **60 isolation trees**. Each radial line represents one tree, while the outer circle represents the maximum depth limit.
The **red lines** represent the **depth** in the trees of the anomalous point, while the **blue lines** represent the **depths** of the normal point.

Clearly as can be seen, on average, the blue lines achieve a much larger radius than the red ones.
By creating many isolation trees, we can use the average depths of the branches to assign anomaly scores to each training instance.





https://arxiv.org/pdf/1811.02141.pdf

Data Pre-processing for Scikit Learn

- Dealing with Outliers
- **Dealing with Missing Values**
- Handling Categorical Data
- Scaling Data
- Handling Imbalance
- Feature Selection

Dealing with Missing Values

- It is common in real-world applications that some features are missing one or more values for various reasons. There could be an **error in the data collection process**, certain fields may have been **left blank in a survey**, **power outage**, etc.

- Most ML algorithms are unable to process these missing values and as such it is important that we deal with all missing values within the data before sending it to the ML algorithm.

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3), index=['a', 'b', 'c'])
seriesB = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(3), index=['b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})
print (df)
print (df.isnull().sum())
```

```
        one     three       two
a  0.376060       NaN  0.111221
b  0.400020  0.164076  0.548106
c  0.507972  0.325337  0.137571
d       NaN  0.823270  0.816618


one      1
three    1
two      0
dtype: int64
```

Notice the **.sum()** method will count the number of missing values in each column.

Dealing with Missing Values

- Please note that the call to **df.isnull().sum()** will only identify the missing values that are specified as **NaN**.

- Missing values may also be represented using dummy values such as ‘?’, -1 ,-99,-999 in your dataset. Missing values such as these will not show up using isnull(). You should be able to identify these in your data exploration stage or when looking for outliers.

- Some of the methods we look at will allow you to specify how a missing values is presented in your dataset.
- However, the first method we will look at will only consider NaN value therefore, you may have to encode your missing values as NaN values.

- A simple way of dealing with this is by **replacing the non-standard missing values with NaN** using df.replace.

Dealing with Missing Values

```
import pandas as pd
import numpy as np

seriesA = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesB = pd.Series([12, 4, 3, '?'], index=['a', 'b', 'c', 'd'])
seriesC = pd.Series([8, 1, '?', 43], index=['a', 'b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                  'three' : seriesC})

df = df.replace('?', np.NaN)

print (df)
print (df.isnull().sum())
```

```
   one  two  three
a  12   12    8
b   4    4    1
c   3    3    ?
d   ?    ?   43

  one    three    two
a  12      8       12
b   4      1        4
c   3     NaN       3
d   NaN    43     NaN

one      1
three    1
two      1
dtype: int64
```

In this case we use replace to replace any occurrence of '?' within the dataframe with NaN.

Dealing with Missing Values - Removal

- One of the easiest ways to deal with missing values (although not always the most appropriate) is to simply remove the corresponding features (columns) or rows from the dataset entirely.
  - <u>**Rows**</u>
    - **df.dropna()** will remove any rows that contain a missing value.
    - **df.dropna(subset=[‘A’])** only drop rows where missing values appear in a specific column in this case column A.
    - **df.dropna(thresh=3)** the parameter thresh specifies the number of <u>non-NAN</u> values that a row must have in order to be retained.
  - <u>**Columns**</u>
    - **df.dropna(axis = 1)** will drop **columns** that have at least one missing value
    - if you want to drop a column of a specific name you can call **df.drop([‘ColumnName'], axis=1)**

- <u>**Each off the above will return a separate copy of the dataset with the new changes (if you want the changes to take place on the current dataframe then you can se the parameter inplace=True)**</u>

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3), index=['a', 'b', 'c'])
seriesB = pd.Series(np.random.rand(4), index=['a', 'b', 'c', 'd'])
seriesC = pd.Series(np.random.rand(3), index=['b', 'c', 'd'])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                 'three' : seriesC})


print (df)
print
newdf = df.dropna()
print (newdf)
```

```
  one             three     two
a 0.867059  NaN           0.255192
b 0.722719  0.420534  0.212348
c 0.328197  0.141678  0.237098
d       NaN     0.458063  0.503182


      one        three     two
b 0.722719 0.420534 0.212348
c 0.328197 0.141678 0.237098
```

Here we drop any rows from our dataframe that contain a missing value.

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(5))
seriesD = pd.Series(np.random.rand(7))

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                  'three' : seriesC,
                  'four' : seriesD})
print (df)
df = df.dropna(thresh=4, axis=1 )
print (df)
```

```
       four       one     three       two
0  0.766476  0.909878  0.476737  0.084872
1  0.810370  0.285238  0.386073  0.268438
2  0.567595  0.162616  0.213230  0.389272
3  0.958997       NaN  0.579480  0.689228
4  0.141135       NaN  0.986758       NaN
5  0.612904       NaN       NaN       NaN
6  0.193091       NaN       NaN       NaN
```

```
import pandas as pd
import numpy as np

seriesA = pd.Series(np.random.rand(3))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(5))
seriesD = pd.Series(np.random.rand(7))

df = pd.DataFrame({'one' : seriesA,
                    'two' : seriesB,
                  'three' : seriesC,
                  'four' : seriesD})
print (df)
df = df.dropna(thresh=4, axis=1 )
print (df)
```

```
       four       one      three      two
0  0.766476  0.909878  0.476737  0.084872
1  0.810370  0.285238  0.386073  0.268438
2  0.567595  0.162616  0.213230  0.389272
3  0.958997       NaN  0.579480  0.689228
4  0.141135       NaN  0.986758       NaN
5  0.612904       NaN       NaN       NaN
6  0.193091       NaN       NaN       NaN

       four     three       two
0  0.766476  0.476737  0.084872
1  0.810370  0.386073  0.268438
2  0.567595  0.213230  0.389272
3  0.958997  0.579480  0.689228
4  0.141135  0.986758       NaN
5  0.612904       NaN       NaN
6  0.193091       NaN       NaN
```

Notice above the threshold value is 4. Therefore, a column must have four or more non-NA values in order to be retained.

Dealing with Missing Values

- Although the removal of missing values may seem convenient it also comes with **significant disadvantages**.

- If you delete a row that has a missing value then you could potentially be **deleting useful feature information**.

- In many cases you may not have an abundance of data and we may end up removing many samples which could in turn **reduce the accuracy** of our model.

- An alternative (which is typically more preferable) to use **imputation** techniques to estimate the missing values from the other training samples in our dataset.
- One of the most common imputation techniques is **mean imputation** where we replace a missing value with the mean of the data items in that column.

Dealing with Missing Values

- Scikit learn provides an easy way of applying this method by using the SimpleImputer class.

- When creating a SimpleImputer object we can specify the **strategy** used for imputing missing values. The most typical is **strategy = mean**, however you can also impute by specifying **strategy = median, strategy = most_frequent** (mode) **or stategy= constant**.

- The most_frequent strategy replaces the missing value by the most frequent values. This is a strategy that is useful to use for **categorical** features.

- Another feature of the SimpleImputer is that we can **specify the missing value** using the parameter **missing_values** (np.nan by default).

```
from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

print (df)

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit(df)

allValues = imputer.transform(df)

print (allValues)
```

```
        one        three       two
a   0.030666      NaN       0.921680
b   0.351147   0.740355     0.478344
c   0.449259   0.299911     0.937952
d      NaN     0.897354     0.168368

[[ 0.03066623  0.64587346  0.92168033]
 [ 0.3511469   0.7403552   0.47834361]
 [ 0.4492592   0.29991101  0.93795242]
 [ 0.27702411  0.89735416  0.16836778]]
```

In this case we can impute for an entire dataset by passing in a pandas **dataframe** (we could also pass in a **NumPy** array).

The array we pass in must be a numerical array. The transform operation will always **returns a NumPy array**.

```
from sklearn.impute import SimpleImputer
# note we are using the dataframe we defined at the start of this section

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')

imputer.fit(df)

allValues = imputer.transform(df)

print (allValues)

df = pd.DataFrame(data = allValues, columns = df.columns)

print(df)
```

```
[[ 0.13159694  0.57903764  0.39386208]
 [ 0.73002787  0.92954245  0.16350405]
 [ 0.67807006  0.58945908  0.60437333]
 [ 0.51323162  0.21811138  0.91419672]]

        one     three       two
0  0.131597  0.579038  0.393862
1  0.730028  0.929542  0.163504
2  0.678070  0.589459  0.604373
3  0.513232  0.218111  0.914197
```

Notice that above we **convert the NumPy array back into a Pandas dataframe** using **pd.DataFrame**

```
import pandas as pd
import numpy as np
from sklearn.impute import SimpleImputer


seriesA = pd.Series(np.random.rand(5))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(["Blue", "Blue", "Yellow", "Brown"])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                        'three' : seriesC})

print (df)
```

```
        one       two   three
0  0.418927  0.734337    Blue
1  0.589525  0.347585    Blue
2  0.863817  0.297105  Yellow
3  0.002752  0.101646   Brown
4  0.745341       NaN     NaN
```

If we have a dataset with heterogenous data types then we may want to apply imputers with **different strategies**.

```
## continued from previous slide

imputer = SimpleImputer(missing_values = np.NaN, strategy='mean')
imputer.fit( df[['two']] )
df['two'] = imputer.transform(  df[['two']]  )

imputer = SimpleImputer(missing_values = np.NaN, strategy="most_frequent")
imputer.fit( df[['three']] )
df['three'] = imputer.transform(  df[['three']]  )

print (df)
```

```
        one       two    three
0  0.418927  0.734337     Blue
1  0.589525  0.347585     Blue
2  0.863817  0.297105   Yellow
3  0.002752  0.101646    Brown
4  0.745341  0.370168     Blue
```

Using ColumnTransformer

- This ColumnTransformer is a class that allows **different columns** of the input to be **transformed separately** and the results combined into a single feature space. This is useful when dealing with datasets that contain heterogeneous data types.

Using ColumnTransformer

- This ColumnTransformer is a class that allows **different columns** of the input to be **transformed separately** and the results combined into a single feature space. This is useful when dealing with datasets that contain heterogeneous data types.

```
import pandas as pd
import numpy as np
from sklearn.compose import ColumnTransformer
from sklearn.impute import SimpleImputer


seriesA = pd.Series(np.random.rand(5))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(["Blue", "Blue", "Yellow", "Brown"])

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                          'three' : seriesC})

print (df)
```

```
        one       two    three
0  0.228164  0.142724   Blue
1  0.940547  0.933032   Blue
2  0.952018  0.475865   Yellow
3  0.185495  0.247130   Brown
4  0.730463       NaN    NaN
```

Using ColumnTransformer

- The main parameter in ColumnTransformer is called **transformers**, which is a list of tuples **(name, transformer, column(s))** specifying the transformer objects to be applied to subsets of the data. See here for a full list of supported transformers.

```
categoricalFeature = ['three']
numericalFeature = ['two']


# On creating the ColumnTransformer we specify a list of transformer operations
preprocessor = ColumnTransformer(transformers=
   [ ('num', SimpleImputer(strategy='mean'), numericalFeature),
    ('cat', SimpleImputer(strategy="most_frequent"), categoricalFeature) ]
,remainder='passthrough')


transformedData = preprocessor.fit_transform(df)
df = pd.DataFrame(data= transformedData)

print (df)
```

```
          0        1         2
0  0.142724   Blue    0.228164
1  0.933032   Blue    0.940547
2  0.475865   Yellow  0.952018
3   0.24713   Brown   0.185495
4  0.449688   Blue    0.730463
```

Multivariate Feature Imputation

- A new (and very welcome) addition to Scikit Learn is a method of multi-variate feature imputation called IterativeImputer.

- It builds a separate model that takes in a **set of features** from the dataset and uses those to **predict the values** for the feature with the **missing values**.

- At each step IterativeImputer will:
  - **Select a feature column** as the target output y and the other feature columns are treated as inputs features X.
  - A **regression model** is fit on (X, y) for known y.
  - Then, the regression model is used to **predict the missing values of y**. This is done for each feature in an iterative fashion.
- Then is repeated for max_iter imputation rounds.

Note from Scikit Learn Website

This estimator is still **experimental** for now: the predictions and the API might change without any deprecation cycle. To use it, you need to explicitly import enable_iterative_imputer.

	one	two	three
0	**0.883211**	**0.281865**	**0.604766**
1	**0.072465**	**0.395314**	**0.297758**
2	**0.958505**	**0.757694**	**0.001842**
3	**0.923956**	**0.373335**	**NaN**
4	**0.570394**	**NaN**	**NaN**

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	**0.5680**
4	0.570394	**0. 73472**	**0.5680**

1. **Select a feature column** as the target output y and the other feature columns are treated as inputs features X.
2. We select the feature with the least number of missing values, which is this case is feature 2.
3. Therefore, feature one and three now becomes the features and two becomes the regression target.

We can only take the rows where there is a valid existing value for column 2.

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	NaN	NaN

	one	three
0	0.883211	0.604766
1	0.072465	0.297758
2	0.958505	0.001842
3	0.923956	**0.5680**

	two
0	0.281865
1	0.395314
2	0.757694
3	0.373335

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	**0.5680**
4	0.570394	**0. 73472**	**0.5680**

Notice, we only take those rows where these isn't a missing value in column 2. (Remember we said previously that we build a **regression model** on (X, y) for known y)

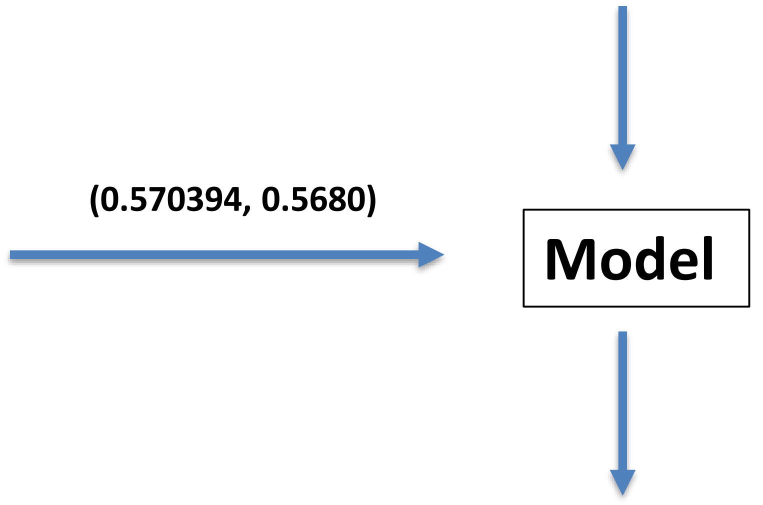
	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	NaN	NaN

Build model for predicting missing values in column two using the other columns as the feature

	one	three
0	0.883211	0.604766
1	0.072465	0.297758
2	0.958505	0.001842
3	0.923956	**0.5680**

	two
0	0.281865
1	0.395314
2	0.757694
3	0.373335

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	0.5680
4	**0.570394**	0. 73472	**0.5680**



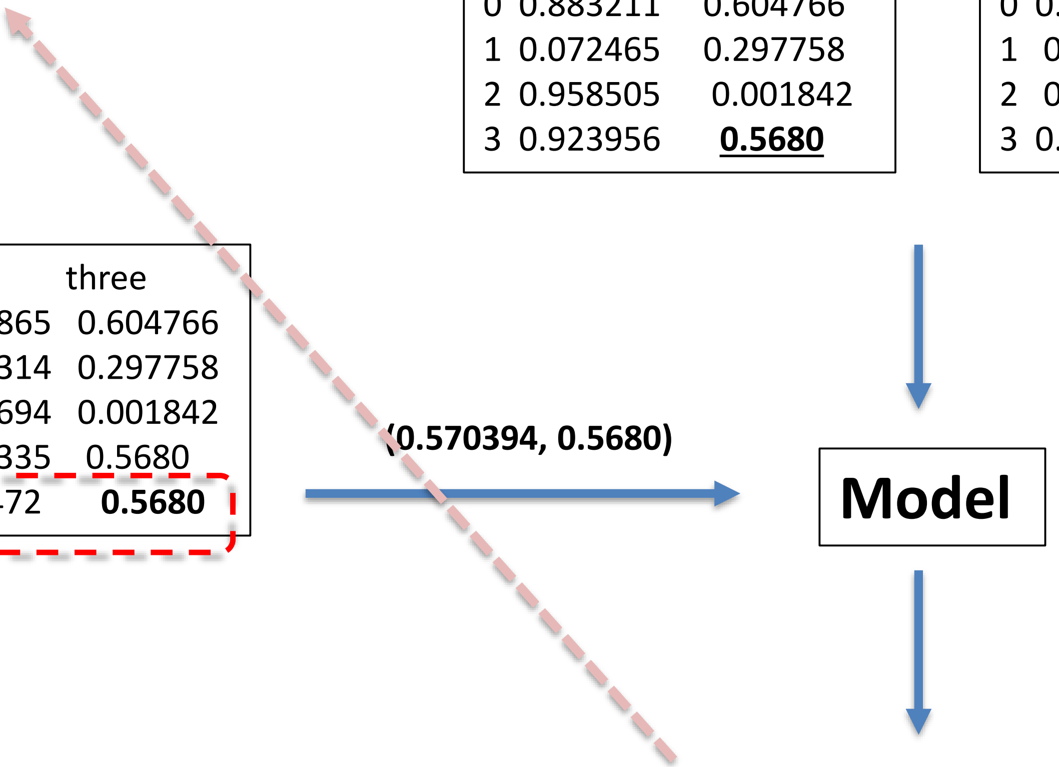
Predict missing values for column 2

	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	NaN
4	0.570394	**NaN**	NaN

Build model for predicting missing values in column two using the other columns as the feature

	one	three
0	0.883211	0.604766
1	0.072465	0.297758
2	0.958505	0.001842
3	0.923956	**0.5680**

	two
0	0.281865
1	0.395314
2	0.757694
3	0.373335



	one	two	three
0	0.883211	0.281865	0.604766
1	0.072465	0.395314	0.297758
2	0.958505	0.757694	0.001842
3	0.923956	0.373335	0.5680
4	**0.570394**	0. 73472	**0.5680**

Predict missing values for column 2

We replace the NaN value for feature 2 with the predicted vaue. This process continues we will build a model for feature 3 next. It is then repeated all over again for a certain number of iterations (10 by default)

```
import pandas as pd
import numpy as np
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer

np.random.seed(0)
seriesA = pd.Series(np.random.rand(5))
seriesB = pd.Series(np.random.rand(4))
seriesC = pd.Series(np.random.rand(3))

df = pd.DataFrame({'one' : seriesA,
                   'two' : seriesB,
                         'three' : seriesC})

print (df)



imp = IterativeImputer(random_state=0)
allValues = imp.fit_transform(df)

df = pd.DataFrame(data = allValues, columns = df.columns)

print(df)
```

```
        one       two     three
0  0.548814  0.645894  0.383442
1  0.715189  0.437587  0.791725
2  0.602763  0.891773  0.528895
3  0.544883  0.963663       NaN
4  0.423655       NaN       NaN

        one       two     three
0  0.548814  0.645894  0.383442
1  0.715189  0.437587  0.791725
2  0.602763  0.891773  0.528895
3  0.544883  0.963663  0.472460
4  0.423655  0.797768  0.482262
```

Missing Values - Advice

- Typically if there is **50%** or more missing values from a feature, then that feature would commonly be **removed** entirely.
- One alternative to deleting a feature that suffers from such a large number of missing values is to create a new **separate binary feature** that indicates if there is a missing value in the original column or not.
- This approach can be useful if the reason for the original missing value has some relationship to the target variable.
- For example, if the missing feature may contain some sensitive personal information, a persons willingness to provide this data may tell us something about that person that might be related to the final target class.
- Typically the binary feature replaces the original feature.

Missing Values - Advice

- As we mentioned previously, **deleting instances (rows)** that contains one or more missing values can result in a very large amount of data being lost. Unless you have an **abundance of data** and a relatively **small amount of missing values** I would not advise employing brute force deletion of rows.

- Imputation is a useful and widely used technique but it is not advisable when features have a very large number of missing values.

- A commonly observed rule is that **imputation** is not advisable on any feature that has **30% or more missing values** and should never be used on a feature that has **50%** or more missing values.

Data Pre-processing for Scikit Learn

- Dealing with Outliers (Optional)
- Dealing with Missing Values
- **Handling Categorical Data**
- Scaling Data
- Handling Imbalance
- Feature Selection
- Dimensionality Reduction

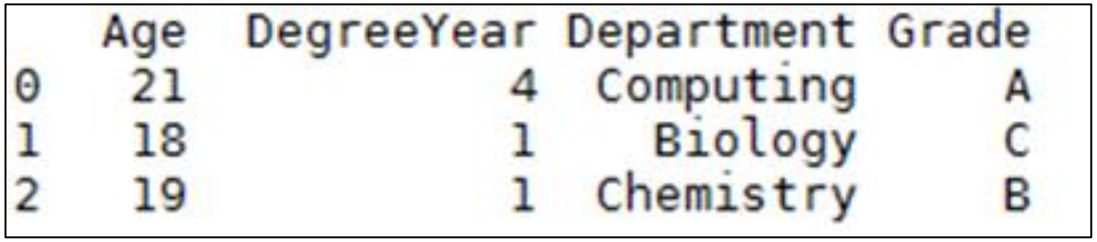
Encoding Categorical Features

- When dealing with **categorical** data, it is important to distinguish between **nominal** and **ordinal** values.

- <u>Ordinal</u> values are variables that have a **logical ordering**. For example, a letter grade for a student, ‘A’, ‘B’, ‘C’, …

- In contrast <u>nominal</u> features have **no inherent ordering**. For example, a features that report’s the colour of a car ‘Red’, ‘Blue’, etc.

- In the example below we have two categorical features. Department being a nominal feature and Grade being an ordinal feature



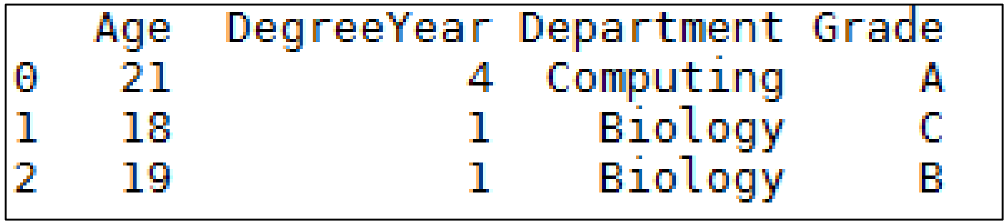
```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1  Chemistry     B
```

Ordinal Variables

- To make sure that our learning algorithms interpret the ordinal features correctly we need to convert the **categorical string values into integers**.
- Unfortunately, the ordering of ordinal values is typically **related to the domain** and as such there is no automatic mechanism of encoding this information.
- Therefore, you need to **specify the mapping manually**, which can be time consuming. In the example on the next slide we specify the mapping for the ordinal feature Grades.
- This involves creating a **dictionary** to specify the direct mapping and using a **dataframe** method call **map**.

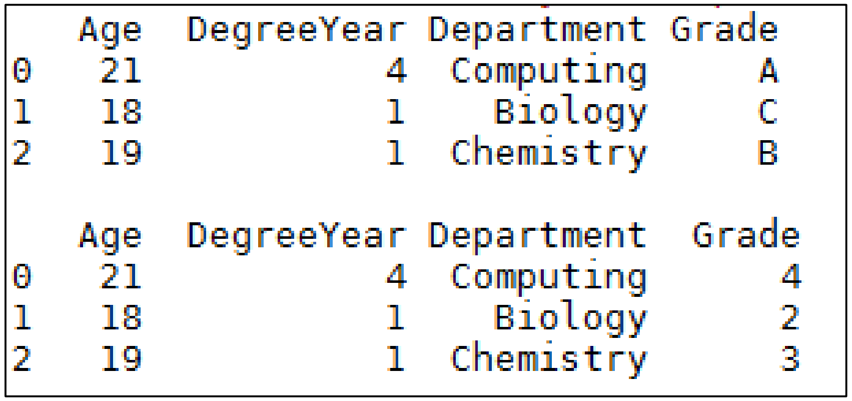
In the following examples we will use the dataframe below

```
import pandas as pd
import numpy as np

seriesA = pd.Series(['A', 'C', 'B'])
seriesB = pd.Series([21, 18, 19])
seriesC = pd.Series([4, 1, 1])
seriesD = pd.Series(['Computing', 'Biology', 'Biology'])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
           'Department' : seriesD})
print (df)
```



```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1    Biology     B
```

```
# continued from previous slide

grade_mapping = {'F':0, 'D':1, 'C':2, 'B':3, 'A':4}

df['Grade'] = df['Grade'].map(grade_mapping)

print (df)
```



```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1  Chemistry     B

   Age  DegreeYear Department  Grade
0   21           4  Computing      4
1   18           1    Biology      2
2   19           1  Chemistry      3
```

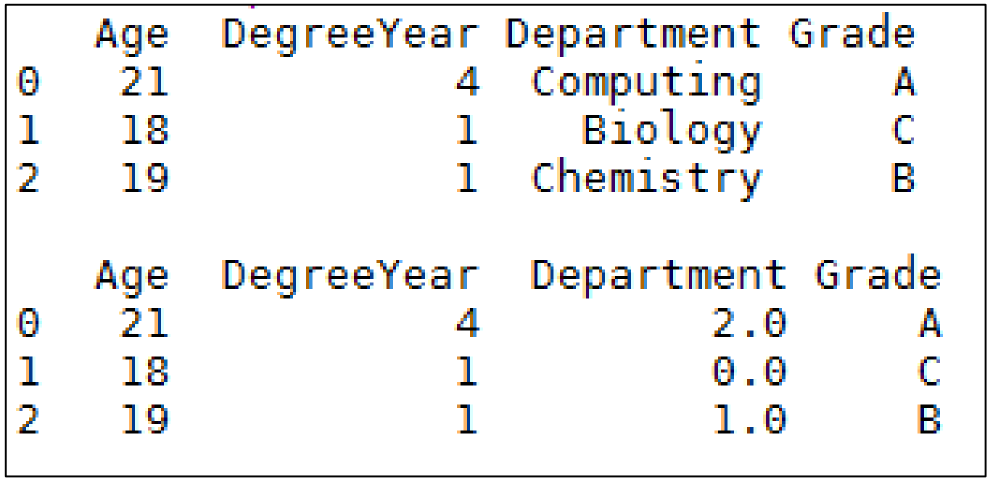
Notice we create a dictionary that provides a mapping from letter grades to numerical values.

We then provide call the map method for the Grade column, which takes the dictionary as an argument.

Nominal Values – Encode as Integer Values

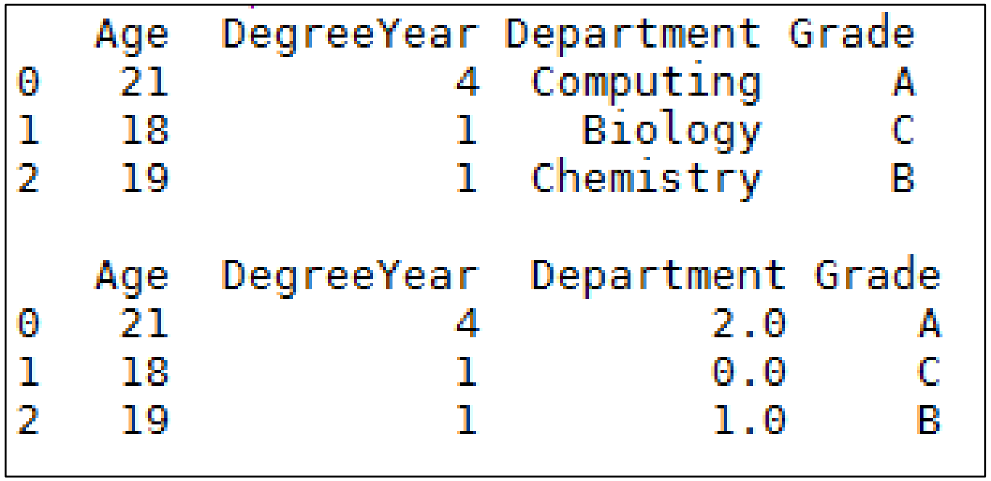
- For nominal values we could directly **encode them into integer values** using the **OrdinalEncoder** from Scikitlearn. After the transform operation the OrdinalEncoder will return an array of numerical values.

```
from sklearn.preprocessing import OrdinalEncoder

enc = OrdinalEncoder()

df["Department"] = enc.fit_transform(df[["Department"]])

print (df)
```



```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1  Chemistry     B

   Age  DegreeYear  Department Grade
0   21           4         2.0     A
1   18           1         0.0     C
2   19           1         1.0     B
```

Encoding Categorical Features – Nominal Values



```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1  Chemistry     B

   Age  DegreeYear  Department Grade
0   21           4         2.0     A
1   18           1         0.0     C
2   19           1         1.0     B
```

- Inadvertently, we have **now specified a ordering** on a nominal categorical feature.
- Although the Department feature has no specific ordering, a learning algorithm will view the encoding as an ordering. Notice that that Computing is closer to Chemistry then it is to Biology.

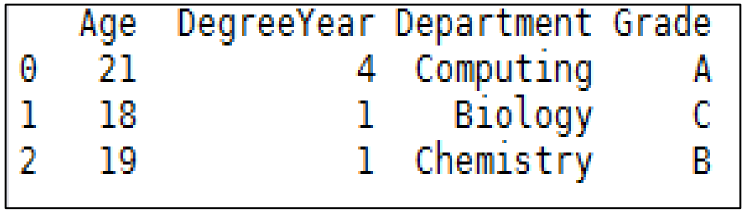
Encoding Categorical Features – Nominal Values

- A common way of addressing this problem is to use a technique referred to as ‘**one-hot encoding**’.

One hot encoding transforms each categorical feature with **n** possible values into **n** binary features

- In the example on the previous slide we could convert the **Department** feature into three new binary features: Computing, Biology and Chemistry.
  - Binary values can then be used indicate the presence of a particular department.
  - For example, a Computing sample would be encoded as Computing = 1, Chemistry = 0 and Biology = 0

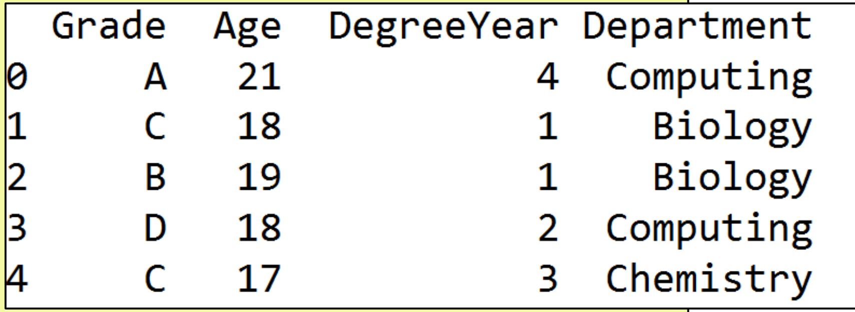
One Hot Encoding



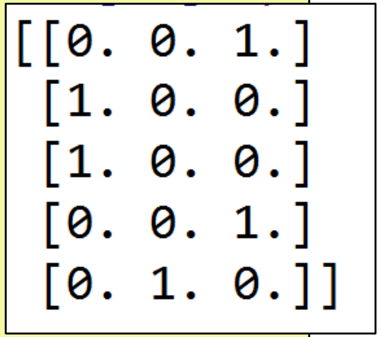
```
   Age  DegreeYear Department Grade
0   21           4  Computing     A
1   18           1    Biology     C
2   19           1  Chemistry     B
```

- **Scikitlearn** provides a **OneHotEncoder** class that allows us to implement a One-Hot Encoder method.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder


seriesA = pd.Series(['A', 'C', 'B', 'D', 'C'])
seriesB = pd.Series([21, 18, 19, 18, 17])
seriesC = pd.Series([4, 1, 1, 2, 3])
seriesD = pd.Series(['Computing', 'Biology', 'Biology', 'Computing', 'Chemistry'])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
            'Department' : seriesD})


encoder = OneHotEncoder(sparse=False)
departmentEncoded =  encoder.fit_transform( df[["Department"]] )

print (departmentEncoded)
```



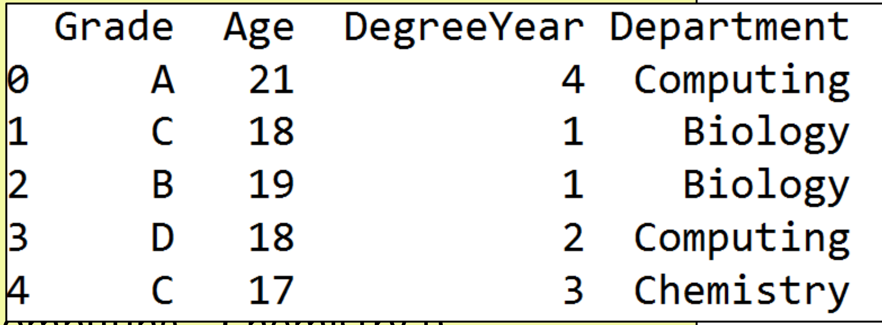
```
  Grade  Age  DegreeYear Department
0     A   21           4  Computing
1     C   18           1    Biology
2     B   19           1    Biology
3     D   18           2  Computing
4     C   17           3  Chemistry
```



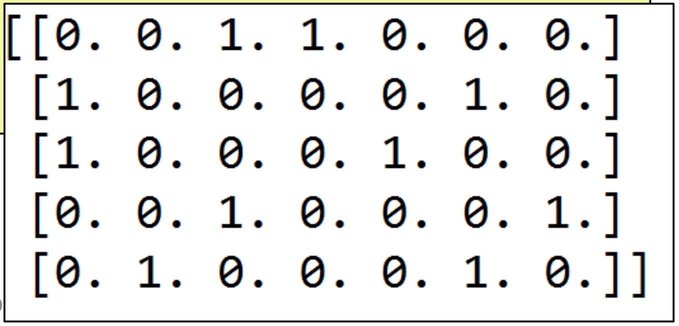
```
[[0. 0. 1.]
 [1. 0. 0.]
 [1. 0. 0.]
 [0. 0. 1.]
 [0. 1. 0.]]
```

Notice when we call fit_transform on the department column it returns the one-hot encoding of this column.

```
import pandas as pd
import numpy as np
from sklearn.preprocessing import OneHotEncoder


seriesA = pd.Series(['A', 'C', 'B', 'D', 'C'])
seriesB = pd.Series([21, 18, 19, 18, 17])
seriesC = pd.Series([4, 1, 1, 2, 3])
seriesD = pd.Series(['Computing', 'Biology', 'Biology', 'Computing', 'Chemistry'])

df = pd.DataFrame({'Grade' : seriesA,  'Age' : seriesB, 'DegreeYear' : seriesC,
            'Department' : seriesD})
print (df)



encoder = OneHotEncoder(sparse=False)
categEncoded = encoder.fit_transform(df[["Department", "Grade"]])


print (categEncoded)
```



```
  Grade  Age  DegreeYear Department
0     A   21           4  Computing
1     C   18           1    Biology
2     B   19           1    Biology
3     D   18           2  Computing
4     C   17           3  Chemistry
```



```
[[0. 0. 1. 1. 0. 0. 0.]
 [1. 0. 0. 0. 0. 1. 0.]
 [1. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 1. 0.]]
```

One-hot Encoding

- The disadvantage of one-hot encoding is that if there are a very **large number of unique values for a categorical feature**, we will consequently end up with a very large number of additional features.

- In turn this can lead to a very long training time or underperformance of some models due to the rapid increase in dimensionality.

- One approach that is used to mitigate the impact of this is dimensionality reduction (techniques such as PCA), which can allow us to in some cases dramatically reduce the overall number of features.