Week 2 lecture notebook

Outline

Missing values

Decision tree classifier

Apply a mask

Imputation

Missing values

In [1]:
```
import numpy as np
import pandas as pd
```

In [2]:
```
df = pd.DataFrame({"feature_1": [0.1,np.NaN,np.NaN,0.4],
                   "feature_2": [1.1,2.2,np.NaN,np.NaN]
                  })
df
```

Out[2]:

	feature_1	feature_2
0	0.1	1.1
1	NaN	2.2
2	NaN	NaN
3	0.4	NaN

Check if each value is missing

In [3]:
```
df.isnull()
```

Out[3]:

	feature_1	feature_2
0	False	False
1	True	False
2	True	True
3	False	True

Check if any values in a row are true

In [4]:
```
df_booleans = pd.DataFrame({"col_1": [True,True,False],
                            "col_2": [True,False,False]
                           })
df_booleans
```

Out[4]:

	col_1	col_2
0	True	True
1	True	False
2	False	False

- If we use pandas.DataFrame.any(), it checks if at least one value in a column is `True` , and if so, returns `True` .
- If all rows are `False` , then it returns `False` for that column

In [5]:
```
df_booleans.any()
```

Out[5]:
```
col_1    True
col_2    True
dtype: bool
```

- Setting the axis to zero also checks if any item in a column is `True`

In [6]:
```
df_booleans.any(axis=0)
```

Out[6]:
```
col_1    True
col_2    True
dtype: bool
```

- Setting the axis to `1` checks if any item in a **row** is `True`, and if so, returns true
- Similarily only when all values in a row are `False`, the function returns `False`.

In [7]:
```
df_booleans.any(axis=1)
```

Out[7]:
```
0     True
1     True
2    False
dtype: bool
```

Sum booleans

In [8]:
```
series_booleans = pd.Series([True,True,False])
series_booleans
```

Out[8]:
```
0     True
1     True
2    False
dtype: bool
```

- When applying `sum` to a series (or list) of booleans, the `sum` function treats `True` as 1 and `False` as zero.

In [9]:
```
sum(series_booleans)
```

Out[9]:
```
2
```

You will make use of these functions in this week's assignment!

This is the end of this practice section.

Please continue on with the lecture videos!

Decision Tree Classifier

In [10]:
```
import pandas as pd
```

In [11]:
```
X = pd.DataFrame({"feature_1":[0,1,2,3]})
y = pd.Series([0,0,1,1])
```

In [12]:
```
X
```

Out[12]:

	feature_1
0	0
1	1
2	2
3	3

In [13]:
```
y
```

Out[13]:
```
0    0
1    0
2    1
3    1
dtype: int64
```

In [14]:
```
from sklearn.tree import DecisionTreeClassifier
```

In [15]:
```
dt = DecisionTreeClassifier()
dt
```

Out[15]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion
='gini',
                       max_depth=None, max_features=None, max_leaf
_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_spl
it=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='depr
ecated',
                       random_state=None, splitter='best')
```

In [16]:
```
dt.fit(X,y)
```

Out[16]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion
='gini',
                       max_depth=None, max_features=None, max_leaf
_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_spl
it=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='depr
ecated',
                       random_state=None, splitter='best')
```

Set tree parameters

In [17]:
```
dt = DecisionTreeClassifier(criterion='entropy',
                            max_depth=10,
                            min_samples_split=2
                           )
dt
```

Out[17]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion
='entropy',
                       max_depth=10, max_features=None, max_leaf_n
odes=None,
                       min_impurity_decrease=0.0, min_impurity_spl
it=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='depr
ecated',
                       random_state=None, splitter='best')
```

Set parameters using a dictionary

- In Python, we can use a dictionary to set parameters of a function.
- We can define the name of the parameter as the 'key', and the value of that parameter as the 'value' for each key-value pair of the dictionary.

In [18]:
```
tree_parameters = {'criterion': 'entropy',
                   'max_depth': 10,
                   'min_samples_split': 2
                  }
```

- We can pass in the dictionary and use `**` to 'unpack' that dictionary's key-value pairs as parameter values for the function.

In [19]:
```
dt = DecisionTreeClassifier(**tree_parameters)
dt
```

Out[19]:
```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion
='entropy',
                       max_depth=10, max_features=None, max_leaf_n
odes=None,
                       min_impurity_decrease=0.0, min_impurity_spl
it=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='depr
ecated',
                       random_state=None, splitter='best')
```

This is the end of this practice section.

Please continue on with the lecture videos!

Apply a mask

Use a 'mask' to filter data of a dataframe

In [20]:
```
import pandas as pd
```

In [21]:
```
df = pd.DataFrame({"feature_1": [0,1,2,3,4]})
df
```

Out[21]:

	feature_1
0	0
1	1
2	2
3	3
4	4

In [22]:
```
mask = df["feature_1"] >= 3
mask
```

Out[22]:
```
0    False
1    False
2    False
3     True
4     True
Name: feature_1, dtype: bool
```

In [23]:
```
df[mask]
```

Out[23]:

	feature_1
3	3
4	4

Combining comparison operators

You'll want to be careful when combining more than one comparison operator, to avoid errors.

- Using the `and` operator on a series will result in a `ValueError`, because it's

In [24]:
```
df["feature_1"] >=2
```

Out[24]:
```
0    False
1    False
2     True
3     True
4     True
Name: feature_1, dtype: bool
```

In [25]:
```
df["feature_1" ] <=3
```

Out[25]:
```
0     True
1     True
2     True
3     True
4    False
Name: feature_1, dtype: bool
```

In [26]:
```
# NOTE: This will result in a ValueError
df["feature_1"] >=2 and df["feature_1" ] <=3
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-26-4feb82af6b46> in <module>
      1 # NOTE: This will result in a ValueError
----> 2 df["feature_1"] >=2 and df["feature_1" ] <=3

/opt/conda/lib/python3.7/site-packages/pandas/core/generic.py in __nonzero__(self)
   1553             "The truth value of a {0} is ambiguous. "
   1554             "Use a.empty, a.bool(), a.item(), a.any() or a.all().".format(
-> 1555                 self.__class__.__name__
   1556             )
   1557         )

ValueError: The truth value of a Series is ambiguous. Use a.empty, a.bool(), a.item(), a.any() or a.all().
```

How to combine two logical operators for Series

What we want is to look at the same row of each of the two series, and compare each pair of items, one row at a time. To do this, use:

- the `&` operator instead of `and`
- the `|` operator instead of `or`.
- Also, you'll need to surround each comparison with parenthese `(...)`

In [27]:
```
# This will compare the series, one row at a time
(df["feature_1"] >=2) & (df["feature_1" ] <=3)
```

Out[27]:
```
0    False
1    False
2     True
3     True
4    False
Name: feature_1, dtype: bool
```

This is the end of this practice section.

Please continue on with the lecture videos!

Imputation

We will use imputation functions provided by scikit-learn. See the scikit-learn [documentation on imputation (https://scikit-learn.org/stable/modules/impute.html#iterative-imputer)](https://scikit-learn.org/stable/modules/impute.html#iterative-imputer)

In [28]:
```
import pandas as pd
import numpy as np
```

In [29]:
```
df = pd.DataFrame({"feature_1": [0,1,2,3,4,5,6,7,8,9,10],
                   "feature_2": [0,np.NaN,20,30,40,50,60,70,80,np.N
aN,100],
                  })
df
```

Out[29]:

	feature_1	feature_2
0	0	0.0
1	1	NaN
2	2	20.0
3	3	30.0
4	4	40.0
5	5	50.0
6	6	60.0
7	7	70.0
8	8	80.0
9	9	NaN
10	10	100.0

Mean imputation

In [30]:
```
from sklearn.impute import SimpleImputer
```

In [31]:
```
mean_imputer = SimpleImputer(missing_values=np.NaN, strategy='mean'
)
mean_imputer
```

Out[31]:
```
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              missing_values=nan, strategy='mean', verbose=0)
```

In [32]:
```
mean_imputer.fit(df)
```

Out[32]:
```
SimpleImputer(add_indicator=False, copy=True, fill_value=None,
              missing_values=nan, strategy='mean', verbose=0)
```

In [33]:
```
nparray_imputed_mean = mean_imputer.transform(df)
nparray_imputed_mean
```

Out[33]:
```
array([[  0.,   0.],
       [  1.,  50.],
       [  2.,  20.],
       [  3.,  30.],
       [  4.,  40.],
       [  5.,  50.],
       [  6.,  60.],
       [  7.,  70.],
       [  8.,  80.],
       [  9.,  50.],
       [ 10., 100.]])
```

Notice how the missing values are replaced with `50` in both cases.

Regression Imputation

In [34]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
```

In [35]:
```
reg_imputer = IterativeImputer()
reg_imputer
```

Out[35]:
```
IterativeImputer(add_indicator=False, estimator=None,
                 imputation_order='ascending', initial_strategy='m
ean',
                 max_iter=10, max_value=None, min_value=None,
                 missing_values=nan, n_nearest_features=None, rand
om_state=None,
                 sample_posterior=False, skip_complete=False, tol=
0.001,
                 verbose=0)
```

In [36]:
```
reg_imputer.fit(df)
```

Out[36]:
```
IterativeImputer(add_indicator=False, estimator=None,
                 imputation_order='ascending', initial_strategy='m
ean',
                 max_iter=10, max_value=None, min_value=None,
                 missing_values=nan, n_nearest_features=None, rand
om_state=None,
                 sample_posterior=False, skip_complete=False, tol=
0.001,
                 verbose=0)
```

In [37]:
```
nparray_imputed_reg = reg_imputer.transform(df)
nparray_imputed_reg
```

Out[37]:
```
array([[  0.,   0.],
       [  1.,  10.],
       [  2.,  20.],
       [  3.,  30.],
       [  4.,  40.],
       [  5.,  50.],
       [  6.,  60.],
       [  7.,  70.],
       [  8.,  80.],
       [  9.,  90.],
       [ 10., 100.]])
```

Notice how the filled in values are replaced with `10` and `90` when using regression imputation. The imputation assumed a linear relationship between feature 1 and feature 2.

This is the end of this practice section.

Please continue on with the lecture videos!

In [ ]:
```

```