Week 2 Jupyter Notebook - Data Preprocessing

0.1 Learning outcomes

1. Dealing with missing data
2. Handling categorical data
3. Selecting meaningful features
4. Assessing feature importance with Random Forests

1 Dealing with missing data

1.1 Identifying missing values in tabular data

Let's create a simple example data frame from a CSV file to get a better grasp of the problem:

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

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,'''


df = pd.read_csv(StringIO(csv_data))
df
```

You can see that two missing cells were replaced by NaN.

We can use the isnull method to return a DataFrame with Boolean values that indicate whether a cell contains a numeric value (False) or if data is missing (True).

[ ]:
```
df.isnull().sum()
```

We can see the number of missing values per column.

1.2 Eliminating samples or features with missing values

The simplest way to deal with missing data is to remove the corresponding features (columns) or samples (rows) from the dataset.

To drop the rows with missing values, we can use dropna method:

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

Similarly, we can drop columns that have at least one NaN in any row:

[ ]:
```
df.dropna(axis=1)
```

We can also try to only drop rows where all columns are NaN

[ ]:
```
df.dropna(how='all')
```

You see the whole array is returned here since we don't have a row with where all values are NaN.

Try to only drop rows where NaN appear in specific columns (here: 'C')

[ ]:
```
df.dropna(subset=['C'])
```

1.3 Imputing missing values

The removal of samples or dropping of feature columns is simple, but we might lose too much valuable data. In this case, we can use different interpolation techniques to estimate the missing values from the other training samples in our datasets.

One common interpolation technique is mean imputation, where we simply replace the missing values with the mean value of the entire feature column.

To achieve this, we can use simpleImputer class from scikit-learn:

[ ]:
```
# again: our original array
df.values
```

[ ]:
```
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
```

Try other options for the strategy parameter 'median' or 'most_frequent'. The latter one replaces the missing values with the most frequent values, which is useful for imputing categorical feature values.

2 Handling categorical data

2.1 Nominal and ordinal features

Let's create a new DataFrame,

```
[ ]: import pandas as pd

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

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

What features are nominal and what features are ordinal?

nominal feature - color; ordinal feature - size; numerical feature - price

2.2 Mapping ordinal features

To make sure the learning algorithm interprets the ordinal features correctly, we need to convert the categorical string values into integers.

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

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

2.3 Encoding class labels

Many machine learning libraries requires 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 achieve this, one way to do is to use the convenient LabelEncoder class directly implemented in scikit-learn.

Note here the fit_transform method is just a shortcut for calling fit and transform separately.

```
[ ]: from sklearn.preprocessing import LabelEncoder
     class_le = LabelEncoder()
     y = class_le.fit_transform(df['classlabel'].values)
     y
```

We can use the inverse_transform method to transform the integer class labels back into their original string representation.

```
[ ]: # reverse mapping
     class_le.inverse_transform(y)
```

2.4 Performing one-hot encoding on nominal features

We could use a similar approach to transform the nominal color column of the dataset:

[ ]:
```
X = df[['color', 'size', 'price']].values

color_le = LabelEncoder()
X[:, 0] = color_le.fit_transform(X[:, 0])
X
```

You can see that the feature ‘color’ is encoded as follows:

`blue = 0, green = 1, red = 2`

If we stop at this point and feed the array to our classifier, we will make a mistake. Because the learning algorithm will assume that green is larger than blue, and red is larger than green. This may result in sub-optimal results.

To solve this issue, we can use a technique called one-hot encoding. It creates a new dummy feature for each unique value in the nominal feature column. We can use the OneHotEncoder that is implemented in scikit-learn’s preprocessing module:

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

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

A more convenient way to create dummy features via one-hot encoding is to use the get_dummies method implemented in pandas.

get_dummies method will only convert string columns and leave all other columns unchanged:

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

Keep in mind that it introduces multicollinearity, which can be an issue for certain methods (e.g. methods that require matrix inversion). If features are highly correlated, matrices are computationally difficult to invert, which can lead to numerically unstable estimates.

To reduce the correlation among variables, we can simply remove one feature column from the one-hot encoded array. By doing so, we do not lose any important information by removing a feature column.

We can drop the first column by passing a True argument to the drop_first parameter:

[ ]:
```
# multicollinearity guard in get_dummies

pd.get_dummies(df[['price', 'color', 'size']], drop_first=True)
```

3 Partitioning a dataset into a seperate training and test set

Let’s prepare a new dataset, the Wine dataset from the UCI.

[ ]:
```python
df_wine = pd.read_csv('https://archive.ics.uci.edu/'
                      'ml/machine-learning-databases/wine/wine.data',
                      header=None)


df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash',
                   'Alcalinity of ash', 'Magnesium', 'Total phenols',
                   'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins',
                   'Color intensity', 'Hue', 'OD280/OD315 of diluted wines',
                   'Proline']

print('Class labels', np.unique(df_wine['Class label']))
df_wine.head()
```

[ ]:
```python
from sklearn.model_selection import train_test_split

X, y = df_wine.iloc[:, 1:].values, df_wine.iloc[:, 0].values

X_train, X_test, y_train, y_test =\
    train_test_split(X, y,
                     test_size=0.3,
                     random_state=0,
                     stratify=y)
```

4 Bringing features onto the same scale

Two common approaches to bring different features onto the same scale:

```
normalization - rescale features to a range of [0, 1]; a special case of min-max scaling
standardization - center the feature columns at mean 0 with standard deviation 1
```

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

mms = MinMaxScaler()
X_train_norm = mms.fit_transform(X_train)
X_test_norm = mms.transform(X_test)
```

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

stdsc = StandardScaler()
X_train_std = stdsc.fit_transform(X_train)
X_test_std = stdsc.transform(X_test)
```

5 Selecting meaningful features

5.1 L1 and L2 regularization as penalties against model complexity

5.2 Sparse solutions with L1-regularization

For regularized models in scikit-learn that support L1 regularization, we can simply set the `penalty` parameter to `'l1'` to obtain a sparse solution:

```
[ ]: from sklearn.linear_model import LogisticRegression
     LogisticRegression(penalty='l1')
```

Apply it to the standardized Wine data: Note that C=1.0 is the default. You can increase or decrease it to make the regulariztion effect stronger or weaker, respectively.

```
[ ]: from sklearn.linear_model import LogisticRegression

     lr = LogisticRegression(penalty='l1', C=1.0, solver='liblinear',␣
      ↪multi_class='ovr')
     lr.fit(X_train_std, y_train)
     print('Training accuracy:', lr.score(X_train_std, y_train))
     print('Test accuracy:', lr.score(X_test_std, y_test))
```

```
[ ]: lr.intercept_
```

```
[ ]: np.set_printoptions(8)
```

```
[ ]: lr.coef_[lr.coef_!=0].shape
```

```
[ ]: lr.coef_
```

Let's vary the regularization strength and plot the regularization path - the weight coefficients of the different features for different reguarization strenghs:

Here C is the inverse of the regularization parameter lambda

```
[ ]: import matplotlib.pyplot as plt

     fig = plt.figure()
     ax = plt.subplot(111)

     colors = ['blue', 'green', 'red', 'cyan',
               'magenta', 'yellow', 'black',
               'pink', 'lightgreen', 'lightblue',
               'gray', 'indigo', 'orange']

     weights, params = [], []
     for c in np.arange(-4., 6.):
         lr = LogisticRegression(penalty='l1', C=10.**c, solver='liblinear',
                                 multi_class='ovr', random_state=0)
```

```
    lr.fit(X_train_std, y_train)
    weights.append(lr.coef_[1])
    params.append(10**c)

weights = np.array(weights)

for column, color in zip(range(weights.shape[1]), colors):
    plt.plot(params, weights[:, column],
             label=df_wine.columns[column + 1],
             color=color)
plt.axhline(0, color='black', linestyle='--', linewidth=3)
plt.xlim([10**(-5), 10**5])
plt.ylabel('weight coefficient')
plt.xlabel('C')
plt.xscale('log')
plt.legend(loc='upper left')
ax.legend(loc='upper center',
          bbox_to_anchor=(1.38, 1.03),
          ncol=1, fancybox=True)
plt.show()
```

What do you observe from the plot?

All feature weights will be zero if we penalize the model with a strong regularization parameter (C<0.1)

6 Assessing feature importance with Random Forests

[ ]:
```
from sklearn.ensemble import RandomForestClassifier

feat_labels = df_wine.columns[1:]

forest = RandomForestClassifier(n_estimators=500,
                                random_state=1)

forest.fit(X_train, y_train)
importances = forest.feature_importances_

indices = np.argsort(importances)[::-1]

for f in range(X_train.shape[1]):
    print("%2d) %-*s %f" % (f + 1, 30,
                            feat_labels[indices[f]],
                            importances[indices[f]]))

plt.title('Feature Importance')
plt.bar(range(X_train.shape[1]),
        importances[indices],
        align='center')

plt.xticks(range(X_train.shape[1]),
           feat_labels[indices], rotation=90)
plt.xlim([-1, X_train.shape[1]])
plt.tight_layout()
plt.show()
```

Disclaimer: The above code is modified from the textbook “Python Machine Learning” by Sebastian Raschka.