BUSA3020-Week5

February 13, 2023

Week 5 Lecture and Computer Lab - Data Preprocessing: Building Good Training Datasets

Unit Convenor & Lecturer

George Milunovich
george.milunovich@mq.edu.au

References

1. Python Machine Learning 3rd Edition by Raschka & Mirjalili - Chapter 4
2. Various open-source material

Learning Objectives

- Removing and Imputing Missing Values from the Dataset
- Getting Categorical Data into Shape for Use with Machine Learning Algorithms
- Selecting Relevant Features for Model Construction
- Feature Importance

Removing and Imputing Missing Values from the Dataset

If the data is inaccurate, missing or irrelevant we will not be able to produce good predictions (no matter how good and sophisticated our forecasting algorithms is) - **Quality** and **amount of useful information** are key determinants of how well a machine learning algorithm can learn - We must examine and preprocess data before we use it to train a machine learning model

Dealing with Missing Data

- Data is often missing
  - Errors in data collection
  - Certain data doesn't exist
  - Surveys are incorrectly answered, e.g. income often understated
  - Etc

One of the main problems is **missing observations**. How Python deals with missing observations: - Common placeholders for missing observations in databases `NaN` - 'not a number' or `NULL` - If we try to train an algorithm using such dataframes we will often get an error - `scikit-learn` was originally developed to work with NumPy arrays - Clean data in pandas and then export into numpy using df.values - Newer versions of some scikit-learn libraries also work with pandas dataframes

Identifying Missing Values in Tabular Data

- When dealing with missing data it is easiest to use pandas
    - isnull method returns a DataFrame with Boolean values to indicate whether a cell contains a numeric value (False) or if data is missing (True)
    - we can also .sum() after isnull which treats False = 0, and True = 1 -> get a number of missing values for each column or row
    - pandas method .info() also provides a count of Non-Null Values for each colum
- Lets create some missing data in a pandas DataFrame

```
import pandas as pd
import numpy as np
from io import StringIO # allows us to read from a string as if we are reading from a file

csv_data = 'A, B, C, D \n 1.0, 2.0, 3.0, 4.0 \n 5.0, 6.0,, 8.0 \n 10.0, 11.0, 12.0,'  # note \n
# print(csv_data, '\n', type(csv_data))

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

```
[ ]:
```

Now we can detect the missing data

```
print('df.info()\n',df.info())
print(2*'\n')

print('df.isnull()\n',df.isnull())
print(2*'\n')

print('df.isnull().sum()\n',df.isnull().sum())
```

```
[ ]:
```

Eliminating Training Examples with Missing Values

- The easiest, but probably **NOT** the best, way to deal with missing data is to remove missing observations (either rows or columns)
- In pandas we can use dropna() method
    - dropna(axis=1) - remove missing features (columns)
    - dropna(axis=0) - remove missing training examples (rows)
    - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dropna.html
    - Most pandas methods have inplace parameter
        * If True, do operation inplace (on our object) and return None, i.e. change directly our dataframe

```
print(df)
print(df.dropna(axis=1)) # drop columns with missing values
print(2*'\n')
print(df)


print(df)
print(df.dropna(axis=0))  # drop rows with missing values
print(2*'\n')
print(df)


print(df)
print(df.dropna(how='all')) # drops rows where all columns are NaN
print(2*'\n')
print(df)

print(df)
print(df.dropna(axis = 0, inplace=True)) # drop rows with missing values and with saves the cha
print(2*'\n')
print(df)
```

[ ]:

[ ]:

[ ]:

[ ]:

Imputing Missing Values

- Deleting missing values is not a good option when we have a small dataset
  - Often we have limited amounts of data and need to preserve as much data as we can
- We may employ different methods of **interpolation** to fill in missing observations
  - Mean/median/mode (most frequent) Imputation
  - E.g. replace missing values with the mean value of the entire feature column

There are a number of libraries that can help deal with missing data 1. `SimpleImputer` class from `scikit-learn` is quite useful - https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html - Can do `mean`, `median`, `most frequent` and `constant` imputation methods by setting `strategy` parameter - `most frequent` method is useful for categorical feature values, e.g. colour names: red, green blue 2. `pandas` has a built in `fillna` method - https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html

Lets interpolate missing values in our dataframe using **mean imputations** using scikit-learn

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

```
df = pd.read_csv(StringIO(csv_data)) # --------- read in data with missing observations again
df

# ---------------- sklearn method -------------------------

imr = SimpleImputer(missing_values = np.nan, strategy='mean') # use mean imputation
imr = imr.fit(df)  # .values is used to export pandas dataframe into numpy array

imputed_data = imr.transform(df)
imputed_data # notice that the data is transferred back into a NumPy array

# ---------------- pandas method -------------------------

df = pd.read_csv(StringIO(csv_data))  # reset df with missing values #  --------- read in data
df

print(df.mean(axis=0))
print(df.fillna(df.mean(axis = 0), inplace=True))  #fill NaN with column mean values
df
```

[ ]:

```
```

[ ]:

```
```

[ ]:

```
```

[ ]:

```
```

Scikit-learn in more depth

- `SimpleImputer` belongs to the **transformer** class in scikit-learn, which are used for data transformation
- Transformers have two main methods
  - `fit` - used to learn parameters, e.g. column means, from training data
  - `transform` - used learned parameters to transform data
- Any data that is to be transformed must have the same number of features as the dataset that was used to fit the transformer

- **Classifiers** that we used so far are similar to **transformers**
  - `fit` method is used to learn parameters
  - `predict` method is used to make predictions using trained parameters

---

Getting Categorical Data into Shape for Use with Machine Learning Algorithms

- So far we have used **numerical** features data

- **Categorical** data
  - A categorical variable is a variable that can take on one of a limited, and usually fixed, number of possible values
  - Each observation is assigned to a particular group or category on the basis of some qualitative property
  - **Ordinal Features**: categorical values that can be ordered or sorted. E.g. shirt size: XL > L > M > S
    - In order to use classifiers on ordinal data properly we need to convert such variables into integers, e.g. XL = 4, L = 3, M = 2, S = 1
  - **Nominal Features**: no ordering possible. E.g. colour: {green, blue, red}
- Create a dataset with
  - a nominal feature (colour)
  - an ordinal feature (size)
  - a numerical feature (price)

```
import pandas as pd

df = pd.DataFrame([
    ['yellow', 'S', 8.2, 'class2'],
    ['green', 'M', 10.1, 'class2'],
    ['red', 'L', 13.5, 'class1'],
    ['blue', 'XL', 15.3, 'class2']])

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

```
[ ]:
```

Mapping Ordinal Features

To map ordinal string features into integers we need to take care of the ordering of the labels
- Often we need to do this manually
- Create a new column whose values reflect ordinal feature labels mapped into integers
- The easiest way is to create a dictionary and map it into the original column

Use `pandas` method `map`
- https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.html
- Used for substituting each value in a Series with another value, that may be derived from a function, a dict or a Series.

Lets consider the example of shirt sizes

```
print(df)

size_mapping = {'XL':4, 'L':3, 'M':2, 'S':1}
print(type(size_mapping))
```

```
df['size2'] = df['size'].map(size_mapping)

df
```

```
[ ]:
```

Tutorial Exercise 1

- To reverse the above mapping we can do:

```
inv_size_mapping = {v:k for k, v in size_mapping.items()}
df['size'].map(inv_size_mapping)
```

Explain what `inv_size_mapping` does by studying the following code.

```
inv_size_mapping = {v:k for k, v in size_mapping.items()}

print(size_mapping)
print(size_mapping.items()) # item returns dictionary's key-value pairs
print(type(inv_size_mapping))

inv_size_mapping_2 = {} # empty dictionary
# print(inv_size_mapping_2.items())

for k, v in size_mapping.items():
#     print(k, v)
    inv_size_mapping_2.update({v:k})

print(size_mapping)
print('inv_size_mapping_2', inv_size_mapping_2)
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

Encoding Class Labels

Although most scikit-learn estimators convert class labels (target variable) to integers internally its best to provide class labels as integer arrays to avoid technical glitches - **Class labels are not ordinal variables** - Doesn't matter which integer we assign to a particular label - It is easiest to enumerate class labels starting at 0

We can use `LabelEncoder` class from scikit-learn for this
- Use `fit_transform` method to encode labels
- Use `inverse_transform` to transform integer labels back to their original string representations

```
from sklearn.preprocessing import LabelEncoder

df

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


y_inverse = class_le.inverse_transform(y)
y_inverse



df['classlabel2'] = y
df['classlabel3'] = y_inverse
df
```

[ ]:

[ ]:

[ ]:

[ ]:

One-Hot Encoding on Nominal Features

- We must be careful not to encode **nominal features** using integer values that represent an ordering
  - E.g. red = 1, blue = 2, green = 3
  - This would confuse our classifier as it will assume that somehow green > blue > red
- Instead we need to create **dummy features (variables)** for each unique value in the nominal feature column
  - There are a number of ways to do this, e.g. use either `OneHotEncoder` scikit-learn libarary or `get_dummies` method from `pandas` -https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html

```
pd.set_option("display.max_rows", None, "display.max_columns", None, "display.width", None) # p
df

one_hot = pd.get_dummies(df[['colour']])
print(one_hot)

df = df.join(one_hot)
df

del df['colour']   # make sure we don't use both 'colour' variables as well as the dummy varia
df
```

[ ]:
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

```

---

## Feature Scaling (Again)

Apart from **decision trees** and **random forests** most machine learning algorithms require feature scaling - Transforming features to the same scale ensures that weights are better optimised

Typically there are two approaches for feature scaling:
1. **Standardization** which we have seen before: if $X \sim (\mu, \sigma) \Rightarrow Z = \frac{X-\mu}{\sigma} \sim (0,1)$ - use `StandardScaler` class from scikit-learn 2. **Normalization** scales features to a range of $[0,1]$ interval - E.g. **min-max scaling** $x_{norm} = \frac{x-x_{min}}{x_{max}-x_{min}}$ - use `MinMaxScalar` class

Which one is more appropriate?
- It really depends on data properties and should be evaluated on a case-by-case basis
- Standardization can be more appropriate for optimization algorithms such as gradient descent - In some cases normalization is more easily interpreted - Can always try both and see which method produces better forecasts

Lets create some data and see how standardization and normalization compare

```
from sklearn.preprocessing import StandardScaler, MinMaxScaler

df = pd.DataFrame(range(0, 6), columns=['X'])
df

standard_scaler = StandardScaler()
df['X_standardized'] = standard_scaler.fit_transform(df['X'].values.reshape(-1,1))

norm_scaler = MinMaxScaler()
df['X_normalized'] = norm_scaler.fit_transform(df['X'].values.reshape(-1,1))

df
```

[ ]:
```

```
[ ]:
```

```
[ ]:
```

```

---

## Selecting Meaningful Features & Overfitting (again)

Typically models perform better on training datasets than on test datasets. Why does this happen?
- Model is overfitted - The model fits to the data in the training dataset too closely, including random variations in the training dataset - The model then tries to predict such random variations in the test dataset but because the variations were random in the first place they don't repeat in the test data
- The possibility of overfitting arises when models are too complex
- Overfitted models are said to have high variance, because the predictions from overfitted models trained on different datasets vary a lot
- We also say that the model does not generalize well to new data
- Poor performance on test data can also happen when the model has not been optimized fully, or the hyperparameters are not selected appropriately

When is a model too complex? - In practice this happens means is that the model has too many parameters - We are considering too many explanatory variables

Possible solutions for overfitting
- Collect more data and train model on larger datasets
- Introduce a penalty for complexity via regularization
- Choose a simpleer model with less parameters
- Reduce the dimension of the data

Next we will discuss - Regularization (again) - Dimensionality reduction via feature selection

Both of these techniques will lead to fewer parameters to be fitted to the data and hence reduce the amount of overfitting.

### L1 and L2 Regularization as Penalties Against Model Complexity

In Week 3 we introduced L2 regularization as a method of reducing model complexity - Large parameter values are penalised in the cost function - L2 uses the values of squared parameters (this designates 2 in L2)

In addition, we can also use L1 regularization which employes absolute values - L2: $||w||_2^2 = \sum_{j=1}^{m} w_j^2 = w_1^2 + w_2^2 + \cdots + w_m^2$ - L1: $||w||_1 = \sum_{j=1}^{m} |w_j| = |w_1| + |w_2| + \cdots + |w_m|$

**Comparison of L1 and L2 regularizations**

- Robustness: L1 is better than L2
  - Robustness is defined as resistance to outliers in a dataset.
  - The more able a model is to ignore extreme values in the data, the more robust it is.
  - L1 norm is **more robust** than the L2 norm
  - L2 norm squares values, so it increases the cost of outliers exponentially and hence tries to make large errors associated with outliers smaller
  - L1 norm only takes the absolute value, so it considers them linearly
- Number of Solutions: L2 has one solution which is better than L1 that has many solutions
  - Because L2 is Euclidean distance, there is always one right answer as to how to get between two points fastest. Because L1 is taxicab distance, there are as many solutions to getting between two points as there are ways of driving between two points in Manhattan! There could be another (mirror image) path of the same length as the blue path.
- Computational difficulty: L2 is easier to optimise than L1

  - L2 has a closed form solution because it's using the square function which can be differentiated
  - L1 does not have a closed form solution because it is a non-differenciable piecewise function, as it involves an absolute value.
  - For this reason, L1 is computationally more expensive, as we can't solve it in terms of matrix math, and most rely on approximations (in the lasso case, coordinate descent).
- Sparsity: L1 is better for feature selection than L2
  - Sparse matrices or sparse arrays are matrices in which most of the elements are zero.
  - By contrast, if most of the elements are nonzero, then the matrix is considered dense.
  - While both L1 and L2 penalties shrink coefficients, L1 tends to shrink coefficients to zero whereas L2 tends to shrink coefficients evenly.
  - L1 is therefore useful for feature selection, as we can drop any variables associated with coefficients that are estimated to be zero.
  - This makes a model robust to potentially irrelevant features in the dataset.
  - L2, on the other hand, is useful when we have multicollinearity.

---

**Impact of Regularization on Logistic Regression Weights**

We'll investigate the impact of regularisation and $C$ (inverse of regularization strength) on Logistic Regression weights

- Import the wine dataset from https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data
  - 178 wine examples
  - 13 features describing their different chemical properties

```
import pandas as pd
import numpy as np

df_wine = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.da

df_wine.columns = ['Class label', 'Alcohol', 'Malic acid', 'Ash', 'Alcalinity of ash', 'Magne
'Flavanoids', 'Nonflavanoid phenols', 'Proanthocyanins', 'Color intensity
'Proline']

df_wine.to_excel('data/wine.xls', index=False)

df_wine = pd.read_excel('data/wine.xls')

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

[ ]:

- Use `train_test_split` libarary to split the data into 70% train and 30% test datasets.
- Stratify according to Class label

```
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=1, strat
```

```
[ ]:
```

- Standardize training and test datasets

```
from sklearn.preprocessing import StandardScaler

stdsc = StandardScaler()

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

```
[ ]:
```

- Normalize training and test datasets (**Normalization** scales features to a range of [0,1] interval)

```
[ ]:
```

- Fit `LogisticRegression` to the training data using L1 penalty (`penalty='l1'`, `C=1.0`, `solver='liblinear'`, `multi_class='ovr'`).
- Compute accuracy for both training and test datasets.

```
from sklearn.linear_model import LogisticRegression

# --- using standardized dataset ---
lr = LogisticRegression(penalty='l1', C=0.1, solver='liblinear', multi_class='ovr')

# Note that C=1.0 is the default. You can increase
# or decrease it to make the regulariztion effect
# stronger or weaker, respectively.

lr.fit(X_train_std, y_train)

print('Training accuracy (standardized):', lr.score(X_train_std, y_train))
print('Testing accuracy (standardized):', lr.score(X_test_std, y_test))


# --- using normalized dataset ---

lr_nor = LogisticRegression(penalty='l1', C=0.1, solver='liblinear', multi_class='ovr')

# Note that C=1.0 is the default. You can increase
# or decrease it to make the regulariztion effect
# stronger or weaker, respectively.
```

```
lr_nor.fit(X_train_nor, y_train)

print('Training accuracy (normalized):', lr_nor.score(X_train_nor, y_train))
print('Testing accuracy (normalized):', lr_nor.score(X_test_nor, y_test))
```

[ ]:

[ ]:

- Print intercepts and weight coefficients of the fitted model

```
print(lr.intercept_)
print(lr.coef_)
print(lr.coef_.shape)

print(lr.coef_[lr.coef_!=0])
print(lr.coef_[lr.coef_!=0].shape)
```

[ ]:

- Vary the regularization strength over the values [0.0001, 0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0, 100000.0]
- Plot the weight coefficients of all features for different regularization strengths

```
# Standardization VS Normalization
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec

fig = plt.figure(figsize=(8,12))
gs = gridspec.GridSpec(2,1)

# standardized data
ax = fig.add_subplot(gs[0])

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

weights, params = [], []
for c in range(-4, 6):
    lr = LogisticRegression(penalty='l1', C=10.**c, solver='liblinear',
                            multi_class='ovr', random_state=0)
    print(10.**c)
    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):
    ax.plot(params, weights[:, column],
             label=df_wine.columns[column + 1],
             color=color)

plt.title("weight coefficients of all features for different regularization strengths (Feature
         fontdict={'weight':'normal','size': 20})
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)

# normalized data
ax = fig.add_subplot(gs[1])

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

weights, params = [], []
for c in range(-4, 6):
    lr_nor = LogisticRegression(penalty='l1', C=10.**c, solver='liblinear',
                            multi_class='ovr', random_state=0)
    print(10.**c)
    lr_nor.fit(X_train_nor, y_train)
    weights.append(lr_nor.coef_[1])
    params.append(10**c)

weights = np.array(weights)

for column, color in zip(range(weights.shape[1]), colors):
    ax.plot(params, weights[:, column],
             label=df_wine.columns[column + 1],
             color=color)

plt.title("weight coefficients of all features for different regularization strengths (Feature
         fontdict={'weight':'normal','size': 20})
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()
```

[ ]:
```

```

[ ]:
```

```

Sequential Feature Selection

An alternative to L1 regularization and sparsity as a method of feature selection is **dimensionality reduction**. There are two main techniques of dimensionality reduction: 1. **Feature Selection** - select a subset of the original features which are relevant to making forecasts 2. **Feature Extraction** - derive information from existing features to construct a new feature subspace - This is done by compressing features in to a new, smaller set of features (will study this next week)

Feature selection problem: Out of an initial set of d features choose k most relevant features where $k < d$.
- E.g. start with $d = 100$ features and reduce them to $k = 25$ most relevant features. - Search is typically done by using **sequential** feature selection algorithms - Sequential selection algorithms are a type of greedy search algorithm

- **Greedy Search Algorithms** - make locally optimal choices at each stage of a combinatorial search problem but generally yeild a suboptimal global solution
  - In the graph below, a greedy algorithm is trying to find the longest path through the graph
    - The number inside each node represents length from previous node
  - To do this, it selects the largest number at each step of the algorithm

  - With a quick visual inspection of the graph, it is clear that this algorithm will not arrive at the correct solution

  - What is the correct solution?
- In contrast **Exhaustive Search Algorithms** - evaluate all possible combinations and are guaranteed to find the optimal solution. However, an exhaustive search is often not computationally feasible.
  - The correct solution for the longest path through the graph is 7,3,1,99. This is clear to us because we can see that no other combination of nodes will come close to a sum of 110.
  - The greedy algorithm fails to solve this problem because it makes decisions purely based on what the best answer at the time is: at each step it did choose the largest number.

Sequential Backward Selection (SBS) is a type of sequential feature selection method - Start by including all possible d features - Sequentially remove features one at a time - In order to determine which feature to remove at each stage define the criterion function J, such as the accuracy of classification - At each step remove a features which results in the least performance loss after its removal

SBS algorithm 1. Initialize the algorithm with $k = d$ 2. Determine the feature, x^- that maximizes the criterion $x^- = \underset{x}{\text{argmax}} J(X_k - x)$ where $x \in X_d$ - Note that $J(X_k - x) < J(X_k)$ meaning that the subset $X_k - x$ has lower accuracy that X_k 3. Remove the feature x^- from the feature set X_k resulting in $X_{k-1} = X_k - x$ - E.g. J = accuracy, $J(x_1, x_2, x_3) = 60\%$, $J(x_1, x_2) = 50\%$, $J(x_1, x_3) = 40\%$, $J(x_2, x_3) = 30\%$ -> choose $J(x_1, x_2) = 50\%$, meaning we eliminated x_3 in this step 4. Terminate if k is equal to the number of desired features, otherwise go to step 2.

SBS Algorithm

- SBS algorithm is not implemented in scikit-learn but we can use it implement the one that comes with the textbook.

```
from sklearn.base import clone
from itertools import combinations
import numpy as np
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split



class SBS():
    def __init__(self, estimator, k_features, scoring=accuracy_score,
                 test_size=0.25, random_state=1):
        self.scoring = scoring
        self.estimator = clone(estimator)
        self.k_features = k_features
        self.test_size = test_size
        self.random_state = random_state

    def fit(self, X, y):

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=self.test_size,
                                     random_state=self.random_state, stratify=y)

        stdsc = StandardScaler()
        X_train = stdsc.fit_transform(X_train)
        X_test = stdsc.transform(X_test)



        dim = X_train.shape[1]
        self.indices_ = tuple(range(dim))
        self.subsets_ = [self.indices_]
        score = self._calc_score(X_train, y_train,
                                 X_test, y_test, self.indices_)
        self.scores_ = [score]

        while dim > self.k_features:
            scores = []
            subsets = []
            for p in combinations(self.indices_, r=dim - 1):
                score = self._calc_score(X_train, y_train,
                                         X_test, y_test, p)
                scores.append(score)
                subsets.append(p)

            best = np.argmax(scores)
            self.indices_ = subsets[best]
            self.subsets_.append(self.indices_)
            dim -= 1

            self.scores_.append(scores[best])
        self.k_score_ = self.scores_[-1]

        return self

    def transform(self, X):
        return X[:, self.indices_]

    def _calc_score(self, X_train, y_train, X_test, y_test, indices):
        self.estimator.fit(X_train[:, indices], y_train)
        y_pred = self.estimator.predict(X_test[:, indices])
        score = self.scoring(y_test, y_pred)
        return score
```

```
[ ]:
```

Feature Selection with SBS algorithm and KNN

We'll use the above SBS algorithm to select a subset of predictors out of 13 features

- Fit the SBS algorithm with KNN on wine training (standarised) data with `k_features = 1`
- Print optimal subsets starting from 1 to 13 features

```
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier

knn = KNeighborsClassifier(n_neighbors=5)

# selecting features
sbs = SBS(knn, k_features=1)

sbs.fit(X, y)

sbs.subsets_
```

```
[ ]:
```

- Plot the classification accuracy of the KNN classifier for each subset of features

```
k_feat = [len(k) for k in sbs.subsets_]

print(k_feat)
print(sbs.scores_)

plt.plot(k_feat, sbs.scores_, marker='o')
plt.ylim([0.7, 1.02])
plt.ylabel('Accuracy')
plt.xlabel('Number of features')
plt.grid()
plt.tight_layout()
# plt.savefig('images/sbs_knn', dpi=300)
plt.show()
```

[ ]:

- Print the smallest feature subset with the classification accuracy of 100%

```
df_scores = pd.DataFrame(sbs.scores_, columns = ['Scores'])
df_scores['Feature Subsets'] = sbs.subsets_
print(df_scores)

smallest_100_subset = list(df_scores['Feature Subsets'].loc[9])

print(smallest_100_subset)

print('features', df_wine.columns[1:]) # start from 1 since 0 is class label

print(df_wine.columns[1:][smallest_100_subset])
```

[ ]:

Train KNN classifier on all features (no feature selection) and evaluate the performance on the test dataset

```
knn.fit(X_train_std, y_train)
print('Training accuracy:', knn.score(X_train_std, y_train))
print('Test accuracy:', knn.score(X_test_std, y_test))
```

[ ]:

Retrain the classifier on the top 4 features selected above and evaluate the performance on the test dataset

```
print(smallest_100_subset)
knn.fit(X_train_std[:, smallest_100_subset], y_train)
print('Training accuracy:', knn.score(X_train_std[:, smallest_100_subset], y_train))
print('Test accuracy:', knn.score(X_test_std[:, smallest_100_subset], y_test))
```

```
[ ]:
```

Assessing Feature Importance with Random Forests

Using a random forest we can measure **feature importance** as the **averaged impurity decrease** computed from all decision trees in the forest - In scikit-learn access feature importance via - `feature_importances` after fitting `RandomForestClassifier` - `SelectFromModel` selects features based on a user-specified importance threshold

A potential problem with feature importance: - If there is multicollinearity, i.e. features are highly correlated, it is difficult to disentangle importance - One feature may rank very high while the other correlated features low (incorrectly) - Can check correlations between the features to see if multicollinearity is a problem in our application - If only interested in predictive ability then this is not a problem as we are not trying to interpret our results

Tutorial Exercise 2: Understand the code below

a) Fit a random forest classifier to the wine data (note: Random Forest doesn't need data to be standardized)

b) Print feature importances

c) Plot feature importances

d) Select features with importance greater than 10%

- Fit a random forest classifier to the wine data (note: Random Forest doesn't need data to be standardized)

```
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)
```

```
[ ]:
```

- Print feature importances

```
importances = forest.feature_importances_

# print(importances)  # prints importances of each feature
# print(np.argsort(importances)) # sort the index of importances from smallest to largest
# print(np.argsort(importances)[::-1]) # reverses the index to get largest to smallest


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

```
# print(X_train.shape)

for f in range(X_train.shape[1]):
#     print(f, indices[f])   # pick up positions from indices
    print(f'{f+1:2})  {feat_labels[indices[f]]:40} {importances[indices[f]]:10}')
```

```
[ ]:
```

- Plot feature importances

```
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.savefig('images/04_09.png', dpi=300)
plt.show()
```

```
[ ]:
```

- Select features with importance greater than 10%

```
from sklearn.feature_selection import SelectFromModel

sfm = SelectFromModel(forest, threshold=0.1, prefit=True)
X_selected = sfm.transform(X_train)
print('Number of features that meet this threshold criterion:', X_selected.shape[1])

for f in range(X_selected.shape[1]):
    print(f'{f + 1:2}) {feat_labels[indices[f]]:40} {importances[indices[f]]:10}')
```

```
[ ]:
```

```
[ ]:
```