hw3

COSC 3337 _Dr. Rizk

1 Problem Set 3

COSC 3337 : Data Science I
Instructor: Nouhad Rizk

Due: June 30 (before 11:59 pm).

How to submit

As mentioned in the lecture, you need to submit the `.ipynb` file with your answers plus an `.html` file, which will serve as a backup for us in case the `.ipynb` file cannot be opened on my or the TA's computer. In addition, you may also export the notebook as PDF and upload it as well.

Again, we will be using the Canvas platform, so you need to submit your homework there. You should be able to resubmit the homework as many times as you like before the due date.

As usual, you do not write the whole code from scratch, and I provided you with a skeleton of code where you need to add the lines that I indicated. Not, however, that everyone's coding style is different. Where I use only one line of code, you may want to use multiple ones. Also, where you use one line of code, I may use multiple ones.

```
[ ]: %load_ext watermark
     %watermark  -d -u -a '<Your Name>' -v -p numpy,scipy,matplotlib,sklearn,mlxtend
```

1.1 1. Hyperparameter Tuning and Model Selection

1.1.1 1.1 [10 pts] Using Grid Search for Hyperparameter Tuning

In this exercise, you will be working with the Breast Cancer Wisconsin dataset, which contains 569 samples of malignant and benign tumor cells.

The first two columns in the dataset store the unique ID numbers of the samples and the corresponding diagnoses (M = malignant, B = benign), respectively. Columns 3-32 contain 30 real-valued features that have been computed from digitized images of the cell nuclei, which can be used to build a model to predict whether a tumor is benign or malignant. The Breast Cancer Wisconsin dataset has been deposited in the UCI Machine Learning Repository, and more detailed information about this dataset can be found at https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic).

The next cell loads the datasets and converts the class label M (malignant) to a integer 1 and the label B (benign) to class label 0.

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL

import pandas as pd


df = pd.read_csv('data/wdbc.data', header=None)

# convert class label "M"->1 and label "B"->0
df[1] = df[1].apply(lambda x: 1 if x == 'M' else 0)


df.head()
```

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL



from sklearn.model_selection import train_test_split



y = df[1].values
X = df.loc[:, 2:].values

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

Now, your task is to use `GridSearchCV` from scikit-learn to find the best parameter for `n_neighbors` of a `KNearestNeighborClassifier`

As hyperparameter values, you only need to consider the number of `n_neighbors` within the range 1-16 (including 16).

[ ]:
```
# MODIFY THIS CELL

from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV



pipe = make_pipeline(# YOUR CODE HERE
                     # YOUR CODE HERE
)

param_grid = [{ # YOUR CODE HERE  }]



gs = GridSearchCV(# YOUR CODE HERE
                  # YOUR CODE HERE
                  iid=False,
                  n_jobs=-1,
                  refit=True,
                  scoring='accuracy',
                  cv=10)

gs.fit(X_train, y_train)

print('Best Accuracy: %.2f%%' % (gs.best_score_*100))
```

Next, print the best parameters obtained from the `GridSearchCV` run and compute the accuracy a `KNearestNeighborClassifier` would achieve with these settings on the test set (`X_test`, `y_test`).

[ ]:
```
# MODIFY THIS CELL

print('Best Params: %s' % # YOUR CODE HERE)
print('Test Accuracy: %.2f%%' % # YOUR CODE HERE)
```

1.1.2 1.2 [10 pts] Estimate the Generalization Performance using the '.632+' Bootstrap

In this exercise, you are asked to compute the accuracy of the model from the previous exercise (1.1) on the test set (`X_test`, `y_test`) using the .632+ Bootstrap method. For this you can use the `bootstrap_point632_score` function implemented in MLxtend for this: http://rasbt.github.io/mlxtend/user_guide/evaluate/bootstrap_point632_score/

- use 200 bootstrap rounds
- set the random seed to 1

The accruacy should be the mean accuracy over the 200 bootstrap values that the `bootstrap_point632_score` method returns.

[ ]:
```
# MODIFY THIS CELL

from mlxtend.evaluate import bootstrap_point632_score
import numpy as np


scores = bootstrap_point632_score(# YOUR CODE HERE)

acc = # YOUR CODE HERE
print('Accuracy: %.2f%%' % (100*acc))
```

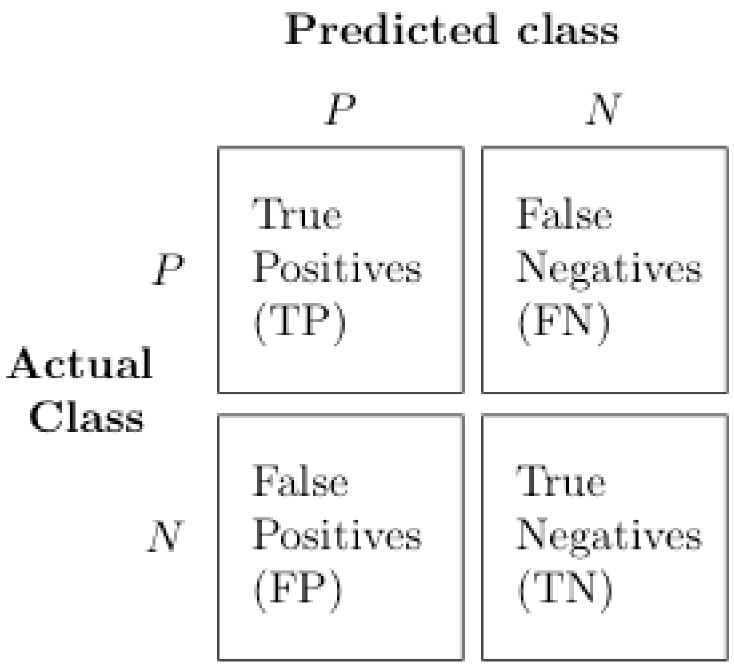
Next, compute the lower and upper bound on the mean accuracy via a 95% confidence interval. For that, you should use the `scores` you computed in the cell above.

[ ]:
```
# MODIFY THIS CELL

lower = # YOUR CODE
upper = # YOUR CODE

print('95%% Confidence interval: [%.2f, %.2f]' % (100*lower, 100*upper))
```

1.2 2. Confusion Matrices

1.2.1 2.1 [10 pts] Contructing a Binary Confusion Matrix

The task of this execise is to construct a binary confusion matrix based of the following form:



Here, assume that the positive class is the class with label 0, and the negative class is the class with label 1. You are given an array of the actual class labels, `y_true`, as well as an array of the predicted class labels, `y_predicted`. The output should be a numpy array, like shown below

```
array([[101, 21],
       [41, 121]])
```

(Note that these number in the array are not the actual, expected or correct values.)

Using the `plot_confusion_matrix` from the `helper.py` script (which should be in the same directory as this notebook) the example array/confusion matrix is visualized as follows:

[ ]:
```
%matplotlib inline
```

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL
```

```python
import numpy as np
from helper import plot_confusion_matrix
import matplotlib.pyplot as plt


example_cm = np.array([[101, 21],
                       [41, 121]])

plot_confusion_matrix(example_cm)
plt.show()
```

Now, your task is to complete the `confusion_matrix_binary` below in order to construct a confusion matrix from 2 label arrays:

- `y_true` (true or actual class labels)
- `y_predicted` (class labels predicted by a classifier)

To make it easier for you, you only need to replace the `???`'s with the right variable name (`tp`, `fn`, `fp`, or `tn`).

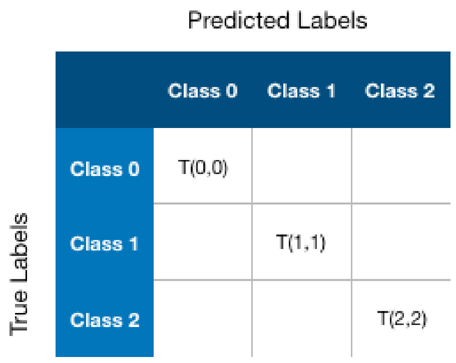
[ ]:
```python
# MODIFY THIS CELL


y_true =      np.array([1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1,␣
 ↪1, 0])
y_predicted = np.array([1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0,␣
 ↪0, 0])


def confusion_matrix_binary(y_true, y_predicted):

    tp, fn, fp, tn = 0, 0, 0, 0

    for i, j in zip(y_true, y_predicted):
        if i == j:
            if i == 0:
                ??? += 1
            else:
                ??? += 1
        else:
            if i == 0:
                ??? += 1
            else:
                ??? += 1

    conf_matrix = np.zeros(4).reshape(2, 2).astype(int)
    conf_matrix[0, 0] = ???
    conf_matrix[0, 1] = ???
    conf_matrix[1, 0] = ???
    conf_matrix[1, 1] = ???

    return conf_matrix

result_matrix = confusion_matrix_binary(y_true, y_predicted)
```

```
[ ]: # EXECUTE BUT DO NOT MODIFY THIS CELL

     print('Conusion matrix array:\n', result_matrix)
```

```
[ ]: # EXECUTE BUT DO NOT MODIFY THIS CELL

     plot_confusion_matrix(result_matrix)
     plt.show()
```

1.2.2 2.2 [10 pts] Constructing a Multiclass Confusion Matrix

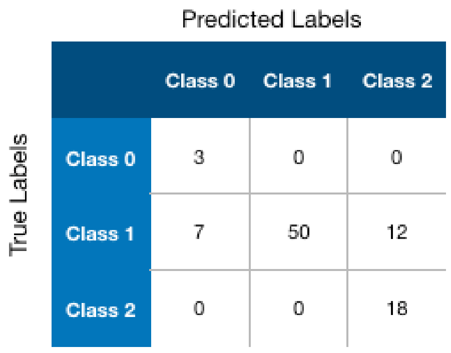
Next, write a version of this confusion matrix that generalizes to multi-class settings as shown in the figure below:



Predicted Labels

True Labels

	Class 0	Class 1	Class 2
Class 0	T(0,0)		
Class 1		T(1,1)	
Class 2			T(2,2)



Predicted Labels

True Labels

	Class 0	Class 1	Class 2
Class 0	3	0	0
Class 1	7	50	12
Class 2	0	0	18

Again, the output should be a 2D NumPy array:

```
array([[3, 0, 0],
       [7, 50, 12],
       [0,  0, 18]])
```

(Note that these number in the array are not the actual, expected or correct values for this exercise.)

There are many different ways to implement a function to construct a multi-class confusion matrix, and in this exercise, you are given the freedom to implement it however way you prefer. Please note though that you should not import confusion matrix code from other packages but implement it by your self in Python (and NumPy).

Note that if there are 5 different class labels (0, ..., 4), then the result should be a 5x5 confusion matrix.

```
[ ]: ## FOR STUDENTS


import numpy as np


def confusion_matrix_multiclass(y_true, y_predicted):

    # YOUR CODE (As many lines of code as you like)

    return matrix


y_true =      [1, 1, 1, 1, 0, 2, 0, 3, 4, 2, 1, 2, 2, 1, 2, 1, 0, 1, 1, 0]
y_predicted = [1, 0, 1, 1, 0, 2, 1, 3, 4, 2, 2, 0, 2, 1, 2, 1, 0, 3, 1, 1]

result_matrix = confusion_matrix_multiclass(y_true, y_predicted)
result_matrix
```

```
[ ]: # EXECUTE BUT DO NOT MODIFY THIS CELL

from helper import plot_confusion_matrix


plot_confusion_matrix(result_matrix)
plt.show()
```

1.2.3 2.3 [10 pts] Binary Confusion Matrices for Multiclass Problems

In this exercise, you will be building binary confusion matrices for multiclass problems as discussed in class when we talked about computing the balanced accuracy. Here, you can reuse the `confusion_matrix_binary` function you implemented in 2.1.

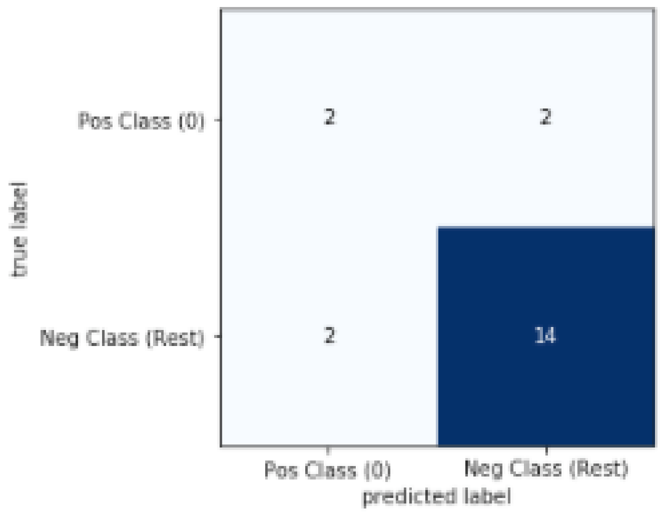
Remember, if we are given 5 class labels (0, ..., 4) then we can construct 5 binary confusion matrices, where each time one of the 5 classes is assigned the positive class where all other classes will be considered as the negative class. The `positive_label` argument in the `binary_cm_from_multiclass` function below can be used to determine which class label refers to the positive class.

Implementing the function below is actually very easy and should only require you to add 2 lines of code with the help of the `np.where` function.

```
[ ]: # MODIFY THIS CELL

def binary_cm_from_multiclass(y_true, y_predicted, positive_label):

    y_true_ary = np.array(y_true)
    y_predicted_ary = np.array(y_predicted)
    y_true_mod = np.where( # YOUR CODE
    y_predicted_mod = np.where( # YOUR CODE

    cm = confusion_matrix_binary(y_true_mod, y_predicted_mod)
    return cm
```

As a hint, the expected output for label 0 as positive label is shown below:

```
Positive Label 0:
 [[ 2  2]
 [ 2 14]]
```



```
[ ]: # EXECUTE BUT DO NOT MODIFY THIS CELL



     y_true =      [1, 1, 1, 1, 0, 2, 0, 3, 4, 2, 1, 2, 2, 1, 2, 1, 0, 1, 1, 0]
     y_predicted = [1, 0, 1, 1, 0, 2, 1, 3, 4, 2, 2, 0, 2, 1, 2, 1, 0, 3, 1, 1]



     mat_pos0 = binary_cm_from_multiclass(y_true, y_predicted, positive_label=0)
     print('Positive Label 0:\n', mat_pos0)

     fig, ax = plot_confusion_matrix(mat_pos0)
     ax.set_xticklabels(['', 'Pos Class (0)', 'Neg Class (Rest)'])
     ax.set_yticklabels(['', 'Pos Class (0)', 'Neg Class (Rest)']);
```

```
[ ]: # EXECUTE BUT DO NOT MODIFY THIS CELL

     mat_pos1 = binary_cm_from_multiclass(y_true, y_predicted, positive_label=1)
     print('\n\nPositive Label 1:\n', mat_pos1)
```

```
fig, ax = plot_confusion_matrix(mat_pos1)
ax.set_xticklabels(['', 'Pos Class (1)', 'Neg Class (Rest)'])
ax.set_yticklabels(['', 'Pos Class (1)', 'Neg Class (Rest)']);

plt.show()
```

1.3 3. [10 pts] Balanced Accuracy

Based on our discussion in class, implement a function that computes the balanced accuracy. You can implement the accuracy whatever way you like using Python and NumPy. Note that you can also re-use the binary confusion matrix code and the `binary_cm_from_multiclass` code if you like (but you don't have to).

Below is a template that you can use that does not require code from the previous exercises (but you can write the function in a different way if you like as long as it gives the correct results).

[ ]:
```
# MODIFY THIS CELL

import numpy as np


def balanced_accuracy(y_true, y_predicted):

    y_true_ary = np.array(y_true)
    y_predicted_ary = np.array(y_predicted)

    unique_labels = np.unique(np.concatenate((y_true_ary, y_predicted_ary)))
    class_accuracies = []
    for l in unique_labels:
        # YOUR CODE HERE
        # YOUR CODE HERE
        # YOUR CODE HERE
        class_accuracies.append(acc)
    return np.mean(class_accuracies)
```

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL

y_targ = [1, 1, 2, 1, 1, 2, 0, 3]
y_pred = [0, 0, 2, 1, 1, 2, 1, 3]

balanced_accuracy(y_targ, y_pred)
```

1.4 4. Receiver Operater Characteristic (ROC)

1.4.1 4.1 [10 pts] Plotting a ROC Curve

In this exercise, you are asked to plot a ROC curve. You are given a 2D array of probability values (`y_probabilities`; see next code cells) where - a value in the first column refer to the probability that a given test example (each row is one test example) belongs to class 0 - a value in the second column refer to the probability that a given test example belongs to class 1

```
[ ]: # EXECUTE BUT DO NOT MODIFY THIS CELL


     from mlxtend.data import iris_data
     from sklearn.model_selection import train_test_split
     from sklearn.linear_model import LogisticRegression


     X, y = iris_data()
     X, y = X[:100, [1]], y[:100]
     X_train, X_test, y_train, y_test = \
         train_test_split(X, y, test_size=0.5, shuffle=True, random_state=0,␣
      ↪stratify=y)

     model = LogisticRegression(solver='lbfgs', random_state=123)
     model.fit(X_train, y_train)

     y_probabilities = model.predict_proba(X_test)

     print(y_probabilities)
```

For this exercise, these scores are probabilities here, but scores can be obtained from an arbitrary classifier (ROC curves are not limited to logistic regression classifiers). For instance, in k-nearest neighbor classifiers, we can consider the fraction of the majority class labels and number of neighbors as the score. In decision tree classifiers, the score can be calculated as the ratio of the majority class labels and number of data points at a given node.

(In case you are curious, 'lbfgs' stands for Limited-memory BFGS, which is an optimization algorithm in the family of quasi-Newton methods that approximates the Broyden–Fletcher–Goldfarb–Shanno; not important to know here though.)

Note: You should only use Python base functions, NumPy, and matplotlib to get full points (do not use other external libraries)

The `pos_label` argument is used to specify the positive label and the threshold. For instance, if we are given score 0.8, this score refers to the "probability" of the positive label. Assuming that the positive label is 1, this refers to a 80% probability that the true class label is 1.

- Note that in the `y_probabilities` array, the second column refers to the probabilities of class label 1.
- The `plot_roc_curve` function should only receive a 1D array for `y_score`. E.g.,

if `y_probabilities` is

```
[[0.44001556 0.55998444]
 [0.69026364 0.30973636]
 [0.31814182 0.68185818]
 [0.56957726 0.43042274]
```

```
[0.86339788 0.13660212]
[0.56957726 0.43042274]
[0.86339788 0.13660212]
[0.44001556 0.55998444]
[0.08899234 0.91100766]
[0.50487831 0.49512169]
[0.74306586 0.25693414]
```

The `y_score` array is expected to be

a) `y_score = [0.5599..., 0.3097..., 0.6818..., 0.4304..., ...]` for `pos_label=1`

and

b) `y_score = [0.4400..., 0.6902..., 0.3181..., 0.5695..., ...]` for `pos_label=0`

[ ]:
```
# MODIFY THIS CELL


import matplotlib.pyplot as plt
import numpy as np


def plot_roc_curve(y_true, y_score, pos_label=1, num_thresholds=100):

    y_true_ary = np.array(y_true)
    y_score_ary = np.array(y_score)
    x_axis_values = []
    y_axis_values = []
    thresholds = np.linspace(0., 1., num_thresholds)

    num_positives = # YOUR CODE
    num_negatives = # YOUR CODE

    for i, thr in enumerate(thresholds):

        binarized_scores = np.where(y_score >= thr, pos_label, int(not␣
 ↪pos_label))

        positive_predictions = # YOUR CODE
        num_true_positives = # YOUR CODE
        num_false_positives = # YOUR CODE

        x_axis_values.append(# YOUR CODE)
        y_axis_values.append(# YOUR CODE)

    plt.step(x_axis_values, y_axis_values, where='post')

    plt.xlim([0., 1.01])
    plt.ylim([0., 1.01])
    plt.ylabel('True Positive Rate')
    plt.xlabel('False Positive Rate')

    return None
```

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL

plot_roc_curve(y_test, y_probabilities[:, 1], pos_label=1)
plt.show()
```

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL

plot_roc_curve(y_test, y_probabilities[:, 0], pos_label=0)
plt.show()
```

1.4.2 4.2 [10 pts] Calculating the ROC AUC

In this exercise, you are asked to modify your previous `plot_roc_curve` function to compute the ROC area under the curve (ROC AUC). To compute the ROC AUC, you can use NumPy's `trapz` function for your convenience (https://docs.scipy.org/doc/numpy-1.15.1/reference/generated/numpy.trapz.html).

- As before, you should only use basic Python functions, NumPy, and matplotlib to get full points for this exercise (do not use other external libraries)

[ ]:
```
# MODIFY THIS CELL


def plot_roc_curve_plus_auc(y_true, y_score, pos_label=1, num_thresholds=100):

    # INSERT YOUR CODE FROM THE PREVIOUS EXERCISE HERE
    # BUT MODIFY IT SUCH THAT IT ALSO RETURNS THE
    # ROC Area Under the Curve
    return roc_auc
```

1) Calculate the ROC AUC for the positive class label 0

[ ]:
```
# DON'T MODIFY BUT EXECUTE THIS CELL TO SHOW YOUR SOLUTION

auc = plot_roc_curve_plus_auc(y_test, y_probabilities[:, 0], pos_label=0)
print('ROC AUC: %.4f' % auc)
```

2) Calculate the ROC AUC for the positive class label 1

[ ]:
```
# DON'T MODIFY BUT EXECUTE THIS CELL TO SHOW YOUR SOLUTION

auc = plot_roc_curve_plus_auc(y_test, y_probabilities[:, 1], pos_label=1)
```

```
print('ROC AUC: %.4f' % auc)
```

1.5 5. Feature Importance

1.5.1 [10 pts] 5.1 Drop-Column Feature Importance

In this exercise, you are asked to implement the "drop-column feature importance" method discussed in class, to measure the importance of individual features present in a dataset.

- You will be using regular accuracy measure as performance metric
- Use 5 fold cross-validation to compute the accuracies

The dataset you will be using for this exercise is the so-called "Wine" dataset.

The Wine dataset is another open-source dataset that is available from the UCI machine learning repository (https://archive.ics.uci.edu/ml/datasets/Wine); it consists of 178 wine samples with 13 features describing their different chemical properties.

The 13 different features in the Wine dataset, describing the chemical properties of the 178 wine samples, are listed in the following table that you will see after executing the next code cell.

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL


import pandas as pd

df_wine = pd.read_csv('data/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']

df_wine.head()
```

The samples belong to one of three different classes, 1, 2, and 3, which refer to the three different types of grape grown in the same region in Italy but derived from different wine cultivars, as described in the dataset summary (https://archive. ics.uci.edu/ml/machine-learning-databases/wine/wine.names).

[ ]:
```
# EXECUTE BUT DO NOT MODIFY THIS CELL


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,
                     stratify=y,
                     random_state=0)
```

Now the task is to implement the `feature_importance_dropcolumn` function to compute the feature importance according the Drop-Column method discussed in class. Here, use the `cross_val_score` function from scikit-learn to compute the acccuracy as the average accuracy from 5-fold cross-validation.

[ ]:
```
# MODIFY THIS CELL


import numpy as np
from sklearn.model_selection import cross_val_score


def feature_importance_dropcolumn(estimator, X, y, cv=5):

    base_accuracy = # YOUR CODE
    column_indices = np.arange(X.shape[1]).astype(int)
    drop_accuracies = np.zeros(column_indices.shape[0])

    for idx in column_indices:
        mask = np.ones(column_indices.shape[0]).astype(bool)
        mask[idx] = False
        drop_accuracy = # YOUR CODE
        drop_accuracies[idx] = # YOUR CODE

    return drop_accuracies
```

Next, apply the `feature_importance_dropcolumn` function to the Wine training dataset (`X_train`, `y_train`) on a `KNeighborsClassifier` (you should use the `make_pipeline` function to create an estimator where the features are scaled to z-scores via the `StandardScaler`, since `KNeighborsClassifier` is very sensitive to feature scales).

- You should use a `KNeighborsClassifier` with 5 nearest neighbors.

[ ]:
```
# MODIFY THIS CELL

from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier




pipe = make_pipeline(
    # YOUR CODE
    # YOUE CODE
)


feature_importance_dropcolumn(# YOUR CODE)
```

1.5.2 [10 pts] 5.2 Random Forest Feature Importance

First, use a `RandomForestClassifier` in your `feature_importance_dropcolumn` from the previous exercise, 5.1. Use a random forest

- with 200 estimators and
- random seed 0.

[ ]:
```
# MODIFY THIS CELL


from sklearn.ensemble import RandomForestClassifier


drop_importances = feature_importance_dropcolumn(
                              # YOUR CODE]
                              X=X_train,
                              y=y_train,
                              cv=5)


print('Drop Importance from RF:', drop_importances)
```

Next, compute the ranking among the features as determined by the outputs of the previous code cell, saved under `drop_importances`. You may use `np.argsort` in your computation, to compute the ranking, where the highest number should correspond to the most important feature.

[ ]:
```
# MODIFY THIS CELL


# YOUR CODE
```

Which are the 3 most important features? You can either write the feature indices below that correspond to the most important features or write out the full column names (you can see the column names in the pandas `DataFrame` in 5.1).

!!! **EDIT THIS CELL TO ENTER YOUR ANSWER** !!!

Next, obtain the feature importance from the random forest classifier directly and compute the ranking as before.

[ ]:
```
# MODIFY THIS CELL

forest = RandomForestClassifier(n_estimators=100, random_state=0)
```

```
forest.fit(X_train, y_train)

print('Random Forest Feature Importance:\n', # YOUR CODE)
```

[ ]:
```
# MODIFY THIS CELL


# YOUR CODE TO RANK THE FEATURES
```

Which are the 3 most important features now? You can either write the feature indices below that correspond to the most important features or write out the full column names (you can see the column names in the pandas `DataFrame` in 5.1).

!!! EDIT THIS CELL TO ENTER YOUR ANSWER !!!

Finally, use the `feature_importance_permutation` function from mlxtend (http://rasbt.github.io/mlxtend/user_guide/evaluate/feature_importance_permutation/) to compute the most important features. Inside `the feature_importance_permutation` function,

- use a random seed of 0
- use 50 permutation rounds

then print the importance values.

[ ]:
```
# MODIFY THIS CELL


from mlxtend.evaluate import feature_importance_permutation


forest = RandomForestClassifier(n_estimators=100,
                                random_state=0)

forest.fit(X_train, y_train)

# YOUR CODE
```

[ ]:
```
# MODIFY THIS CELL


# YOUR CODE TO RANK THE FEATURES
```

Which are the 3 most important features now? You can either write the feature indices below that correspond to the most important features or write out the full column names (you can see the column names in the pandas `DataFrame` in 5.1).

!!! EDIT THIS CELL TO ENTER YOUR ANSWER !!!

1.5.3 [10 pts] 5.3 Creating your Own Feature Selection Transformer Class

This section will help you understand how you can implement your own feature selection method in a way that is compatible with scikit-learn.

The following code (`ColumnSelector`) implements a feature selector that works similarly to the feature selctors implemented in scikit-learn. However, this `ColumnSelector` does not do anything automatically.

[ ]:
```python
# EXECUTE BUT DO NOT EDIT THIS CELL

from sklearn.base import BaseEstimator
import numpy as np


class ColumnSelector(BaseEstimator):

    def __init__(self, cols=None):
        self.cols = cols

    def fit_transform(self, X, y=None):
        return self.transform(X=X, y=y)

    def transform(self, X, y=None):
        feature_subset = X[:, self.cols]
        if len(feature_subset.shape) == 1:
            feature_subset = feature_subset[:, np.newaxis]
        return feature_subset

    def fit(self, X, y=None):
        return self
```

As the name implies, we `ColumnSelector` selects specific columns that we as the user need to specify. For example, consider the Wine dataset from earlier:

[ ]:
```python
# EXECUTE BUT DO NOT EDIT THIS CELL

import pandas as pd

df_wine = pd.read_csv('data/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']

df_wine.head()
```

[ ]:
```
# EXECUTE BUT DO NOT EDIT THIS CELL

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,
                     stratify=y,
                     random_state=0)
```

Via the `ColumnSelector`, we can select select specific columns from the dataset. E.g., to select the 1st, 6th, and 9th column, and 12th column, we can initialize the `ColumnSelector` with the argument `cols=[0, 5, 8, 11]` and use the transform method as shown below:

[ ]:
```
# EXECUTE BUT DO NOT EDIT THIS CELL

col_sele = ColumnSelector(cols=[0, 5, 8, 11])
reduced_subset = col_sele.transform(X_train)

print('Original feature set size:', X_train.shape)
print('Selected feature set size:', reduced_subset.shape)
```

Your task now is to use the `feature_importances_` attribute from a fitted random forest model inside a custom feature selector. Using this feature selector, you should be able to select features as follows:

```
forest = RandomForestClassifier(n_estimators=100, random_state=123)

selector = ImportanceSelector(num_features=3, random_forest_estimator=forest)
selector.fit(X_train, y_train)
reduced_train_features = selector.transform(X_train, y_train)
```

- If `num_features=3` as shown above, this means that we are interested to select the top 3 most important features from a dataset based on the random forest feature importance values.
- Actually, while it might be more interesting to implement a feature selctor based on the column-drop performance (which would then be somewhat related to sequential feature selection), we use the feature importance values from a `RandomForest`'s `feature_importances_` attribute for simplicity here, to allow you to implement this method in case your `feature_importance_dropcolumn` function does not work correctly.

[ ]:
```
# MODIFY THIS CELL

from sklearn.base import BaseEstimator
import numpy as np
```

```
class ImportanceSelector(BaseEstimator):

    def __init__(self, num_features, random_forest_estimator):
        self.num_features = num_features
        self.forest = random_forest_estimator

    def transform(self, X, y=None):

        # Feature by increasing feature importance:
        features_by_importance = # YOUR CODE
        top_k_feature_indices = # YOUR CODE

        feature_subset = X[:, top_k_feature_indices]
        if len(feature_subset.shape) == 1:
            feature_subset = feature_subset[:, np.newaxis]
        return feature_subset

    def fit(self, X, y=None):
        self.forest.fit(X, y)
        return self
```

Now, use the `ImportanceSelector` to select the 3 most important features in the dataset:

```
[ ]: # MODIFY THIS CELL

from sklearn.ensemble import RandomForestClassifier



forest = RandomForestClassifier(n_estimators=100, random_state=123)

selector = # YOUR CODE
# YOUR CODE
reduced_train_features = # YOUR CODE

print('Original feature set size:', X_train.shape)
print('Selected feature set size:', reduced_train_features.shape)
print('First 5 rows:\n', reduced_train_features[:5])
```

1.6 (5 pts) Bonus Exercise: Evaluating a KNN Classifier on Different Feature Subsets

In this *Bonus Exercise*, your task is to use a scikit-learn pipeline to fit a KNN classifier based on different 2-feature combinations and different values of k (number of neighbors) via grid search. More specifically,

1. Create a scikit-learn pipeline that consists of a `StandardScaler`, a `ColumnSelector`, and a `KNeighborsClassifeir` (think about the right way to order these elements in the pipeline);
2. Using this pipeline, find the best value for `k` in the KNN classifier as well as the best feature

combination (restricted to 2-feature subsets for simplicity) using `GridSearchCV`;

3. Fit the best model determined via grid search on the whole training set and evaluate the performance on the test set.

[ ]:
```
# EXECUTE BUT DO NOT EDIT


import pandas as pd


df_wine = pd.read_csv('data/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']

df_wine.head()
```

[ ]:
```
# EXECUTE BUT DO NOT EDIT

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,
                     stratify=y,
                     random_state=0)
```

[ ]:
```
# EXECUTE BUT DO NOT EDIT THIS CELL

from sklearn.base import BaseEstimator
import numpy as np


class ColumnSelector(BaseEstimator):

    def __init__(self, cols=None):
        self.cols = cols

    def fit_transform(self, X, y=None):
        return self.transform(X=X, y=y)
    def transform(self, X, y=None):
        feature_subset = X[:, self.cols]
        if len(feature_subset.shape) == 1:
            feature_subset = feature_subset[:, np.newaxis]
        return feature_subset

    def fit(self, X, y=None):
        return self
```

Modify the following code cell to create a list of all possible 2-feature combinations:

[ ]:
```
# MODIFY THIS CELL

import itertools


all_combin_2 = list(itertools.combinations( # YOUR CODE)


print('Number of all possible 2-feature combinations:', len(all_combin_2))
```

Modify the following code cell to create a `pipeline` (as explained at the beginning of this section), and use the given `param_grid` to fit the `GridSearchCV` to obtain the best parameters settings and a classifier fit to `X_train` and `y_train` based on these best hyperparameter values.

(Note that the code may take 10-30 seconds to execute.)

[ ]:
```
# MODIFY THIS CELL

from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV


pipe = make_pipeline(
# YOUR CODE
# YOUR CODE
# YOUR CODE
)


param_grid = {'kneighborsclassifier__n_neighbors': list(range(1, 8)),
              'columnselector__cols': all_combin_2}

gsearch = GridSearchCV(pipe,
                       param_grid=param_grid,
                       refit=True,
                       iid=False,
                       cv=5)

gsearch.fit(X_train, y_train)
```

[ ]:
```
# EXECUTE BUT DO NOT EDIT


print(gsearch.best_params_)
```

Based on the best combination of a 2-feature subset and the number of `n_neigbors` your model should be fit the the training dataset now. Use the fitted model and compute its classification accuracy on the test set (`X_test`, `y_test`).

[ ]:
```
# MODIFY THIS CELL

# YOUR CODE TO COMPUTE THE TEST ACCURACY
```