03_outlier

December 18, 2024

1 Outlier Detection (optional)

Now that the model is trained, we can take a look at outlier detection.
In this notebook, three different methods to identify outliers are presented, i.e. data-driven choices for distinguishing known from unknown (class not in training data) spectra.

For spectra of N different classes, each of these classes is selected as outlier for one outlier training iteration.
In each iteration, the model is trained with N-1 classes (known spectra) and then tested with 1 class (unknown spectra).
With these results, the best, data-driven discrimination between known and unknown data can be found.

This notebook only serves for exploratory purposes; you need to manually implement/update the final outlier criterion in your measurement pipeline.

The python library used for model training and testing is `tools_outlier.py` and all visualizations are specified in `plotting.py`.
As we did before, we set the main parameters for model training and save them as global variables.

[1]:
```
from tools_outlier import *
from plotting import *

# Set global variables
dets_tr = ['simulated+bg']  # only simulated spectra + backgrounds (default)
GlobalVariables.dets_tr = dets_tr

min_channel_tr = 7
GlobalVariables.min_channel_tr = min_channel_tr

min_scores_norm = 0.1
GlobalVariables.min_scores_norm = min_scores_norm
```

[2]:
```
# load data
data, all_isotopes = load_spectral_data(dir_numpy_ready, dets_tr)
all_isotopes = [x[0] for x in all_isotopes]  # only single-label spectra␣
 ↪allowed for model training
data = remove_empty_or_negative_spectra(data)
```

```
# We need to identify the isotope with the fewest data (for class balance later␣
 ↪on)
list_counts = count_spectra_per_isotope(data)
n_cut = min(list_counts)  # find minimum number
```

```
['Am241']: 814 spectra
['Co60']: 1351 spectra
['Cs137']: 1230 spectra
['Eu152']: 1498 spectra
['background']: 1226 spectra
```

[3]:
```
# We now choose the isotopes that should "play" outlier
# Then, we loop over these isotopes, train the model based on the known␣
 ↪isotopes and output the relevant data to best discriminate known from␣
 ↪unknown isotopes

# choose which isotopes are included in outlier analysis
isotope_outliers = set_isotope_outliers(all_isotopes)  # e.g. isotope_outliers␣
 ↪= ['Am241', 'Co60', 'Cs137'] (without background)

if len(isotope_outliers) == 0:
    print('Error: No outlier isotopes were provided')
    exit()

for i, isotope_outlier in enumerate(isotope_outliers):

    # main function call
    xi_known, xi_unknown = simulate_outlier(isotope_outlier)
    # xi_known contains features of known isotopes (that were used in training)
    # xi_unknown contains features of unknown isotopes (that were not used in␣
 ↪training)

    if i > 0:  # list is already defined, append to array
        x_all_known = np.append(x_all_known, np.array(xi_known), axis=0)

        # take n_cut random entries
        x_all_unknown = np.append(x_all_unknown, np.array(xi_unknown)[np.random.
 ↪choice(np.arange(len(xi_unknown)), size=n_cut, replace=False), :], axis=0)

    else:  # first loop: set up array
        x_all_known   = np.array(xi_known)
        x_all_unknown = np.array(xi_unknown)

# The output "explained variance ratio" measures the quality of the model␣
 ↪training:
# Based on the training data (first number) this value should be close to 100%
```

```
# Based on the unknown data ("outlier") this number should be <100% (usually␣
↪around 10%). It can also be negative.
```

```
 Excluded isotope: ['Am241']
Building loadings from mean spectra of those isotopes: ['Co60' 'Cs137' 'Eu152'
'background']
Explained variance ratio:  89.7%

Explained variance ratio:  5.1%


 Excluded isotope: ['Co60']
Building loadings from mean spectra of those isotopes: ['Am241' 'Cs137' 'Eu152'
'background']
Explained variance ratio:  89.5%

Explained variance ratio: -5.2%


 Excluded isotope: ['Cs137']
Building loadings from mean spectra of those isotopes: ['Am241' 'Co60' 'Eu152'
'background']
Explained variance ratio:  89.1%

Explained variance ratio:  5.1%


 Excluded isotope: ['Eu152']
Building loadings from mean spectra of those isotopes: ['Am241' 'Co60' 'Cs137'
'background']
Explained variance ratio:  97.6%

Explained variance ratio:  30.7%
```

Optional: Set an imbalance factor between known and unknown data Since it is not equally likely to encounter known and unknown spectra, you can set a factor of imbalance below. This will adjust the sizes of the datasets for the further analyses.
In our example, we set `factor_imbalance = 10` for illustrative purposes. The imbalance factor quantifies how much more known than unknown data we have in the training. On average, the final model will hence predict that every `factor_imbalance` spectra is an outlier. A higher value will make the model more sensitive towards outliers, but also creates more false positives.

[4]:
```
factor_imbalance = 10  # here: factor 10 between known and unknown spectra
x_all_known = x_all_known[:factor_imbalance*x_all_unknown.shape[0], :]
```

[5]:
```
# Create the label column to identify known /
known_labels = np.zeros((x_all_known.shape[0], 1))  # 0 for all rows in␣
 ↪x_all_known
unknown_labels = np.ones((x_all_unknown.shape[0], 1))  # 1 for all rows in␣
 ↪x_all_unknown

# Stack the numpy arrays vertically
x_combined = np.vstack((x_all_known, x_all_unknown))

# Stack the label column with the combined data
labels_combined = np.vstack((known_labels, unknown_labels))

# Get the number of columns in x_combined
num_scores = x_combined.shape[1]-5  # Number of columns in x_combined minus␣
 ↪manual values

# Dynamically generate the "scoreX" column names based on the number of columns␣
 ↪in x_combined
score_column_names = [f"score{i+1}" for i in range(num_scores)]  # Adjust based␣
 ↪on how many additional columns are fixed

# Add any fixed column names after the dynamically generated score columns
other_column_names = ['scores_mean', 'scores_median', 'expl.var.', 'cos.sim.',␣
 ↪'scores_norm_abs', 'label']

# Combine dynamically generated score columns with other fixed columns
column_names = score_column_names + other_column_names

check_column_names_match(x_combined, labels_combined, column_names)

# Combine the arrays into a pandas DataFrame
df_combined = pd.DataFrame(np.hstack((x_combined, labels_combined)),␣
 ↪columns=column_names)

# Shuffle data
df_combined = df_combined.sample(frac=1).reset_index(drop=True)

# Check the resulting DataFrame
print("The following information is available and used for the model training␣
 ↪to discriminate known from unknown spectra.")
print("We show the first fews lines as an example:")
print(df_combined.head())
```

```
The following information is available and used for the model training to
discriminate known from unknown spectra.
We show the first fews lines as an example:
     score1    score2    score3  score4  scores_mean  scores_median  \
```

```
0  0.994325  0.004318  0.001357     0.0         0.25       0.002838
1  0.860289  0.126554  0.013158     0.0         0.25       0.069856
2  1.000000  0.000000  0.000000     0.0         0.25       0.000000
3  1.000000  0.000000  0.000000     0.0         0.25       0.000000
4  0.852159  0.147841  0.000000     0.0         0.25       0.073920

   expl.var.  cos.sim.  scores_norm_abs  label
0   0.982479  0.991201         0.994335    0.0
1   0.573948  0.792534         0.869647    0.0
2   0.997358  0.998690         1.000000    0.0
3   0.717872  0.884935         1.000000    0.0
4  -0.194783  0.290947         0.864889    0.0
```

[6]:
```
n_tot = len(df_combined)
f_train = 0.8  # fraction of training data
n_train = int(f_train * n_tot)

# Split the data into a training and a testing set (here, Ntrain is the number␣
 ↪of lines for training)
train_features = np.array(df_combined.iloc[:n_train, :-1])
train_targets = np.array(df_combined.iloc[:n_train,-1])

test_features = np.array(df_combined.iloc[n_train:, :-1])
test_targets = np.array(df_combined.iloc[n_train:,-1])

test_cos_sim = np.array(df_combined["cos.sim."])  # because this will be the␣
 ↪important quantity later
all_targets = np.array(df_combined.iloc[:, -1])
```

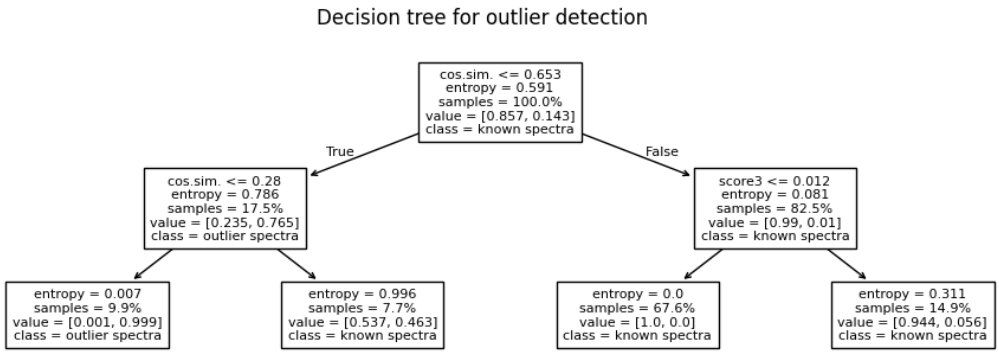
1.1 Step 1: Finding the best feature for outlier identification

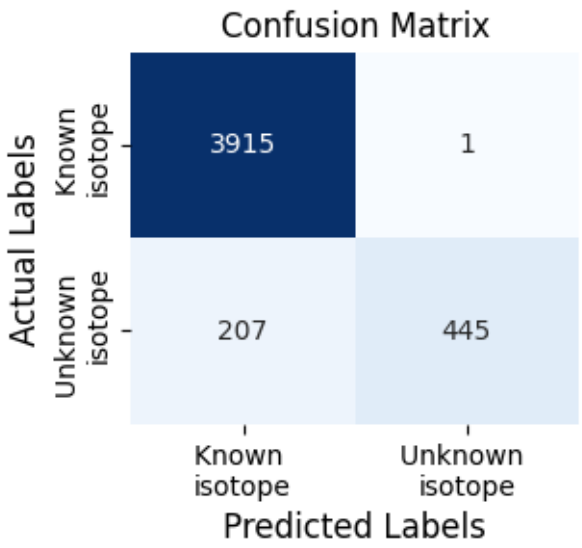
Let's train a decision tree model to predict the best discrimination between known and unknown spectra.
Decision trees are very transparent machine learning models, where the data set is split based on a set of consecutive rules.
How to read the decision tree plot (see below): - The first line of reach cells show the condition based on which a split is performed. - If the condition is TRUE, data goes to the left side. If it is FALSE, data goes to the right side. - values = [known spectra, outlier spectra] shows the number of spectra that are still in a node. - The objective is to get these values as "pure" as possible. - The pureness of the sample is given be the (information) entropy. - The construction algorithm of a decision tree tries to minimize the entropy.

[7]:
```
# Train the model
tree_depth = 2
tree = DecisionTreeClassifier(criterion='entropy', max_depth=tree_depth).
 ↪fit(train_features,train_targets)
```

```
# Predict the classes of new, unseen data
prediction = tree.predict(test_features)

# Check the accuracy
accuracy = accuracy_score(test_targets, prediction)
precision = precision_score(test_targets, prediction)
recall = recall_score(test_targets, prediction)
print(f'{accuracy=:.3f}, {precision=:.3f}, {recall=:.3f}')

# plot tree
feature_names = df_combined.columns[:-1]
fig_tree = plt.figure(figsize=(12, 3.5))
fig_tree.suptitle('Decision tree for outlier detection')
skl.tree.plot_tree(tree,
                   max_depth=tree_depth,
                   feature_names=feature_names,
                   class_names=['known spectra', 'outlier spectra'],
                   proportion=True,
                   fontsize=8)

plot_outlier_confusion(test_targets, prediction)
```

```
accuracy=0.954, precision=0.998, recall=0.683
```

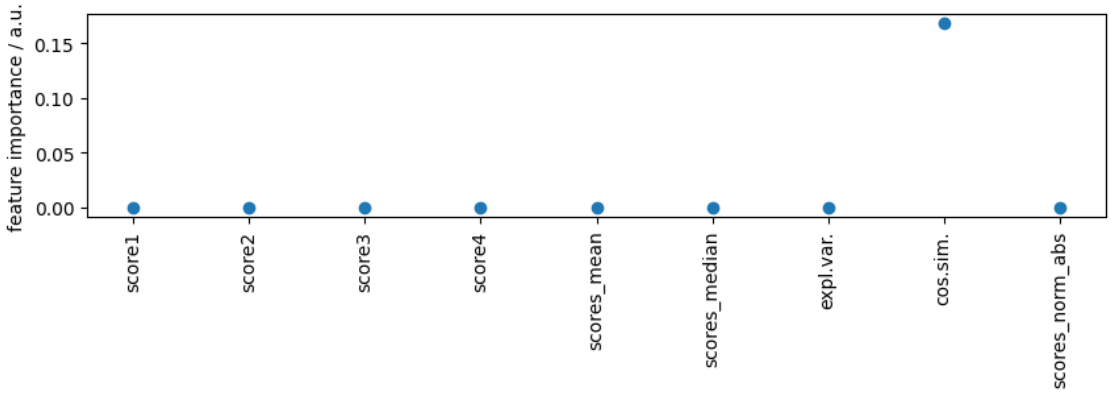




1.1.1 Analysis of Feature Importance

We can analyze the importance of the features to decide which feature will be used to distinguish known from unknown features.
In our example, the cosine similarity is the most important feature.

[8]:
```
feature_importance = permutation_importance(tree, train_features,␣
↪train_targets, n_repeats=5)
y = feature_importance['importances_mean']

plot_feature_importance(feature_names,y)
```



1.2 Step 2: Decision boundary for most important feature

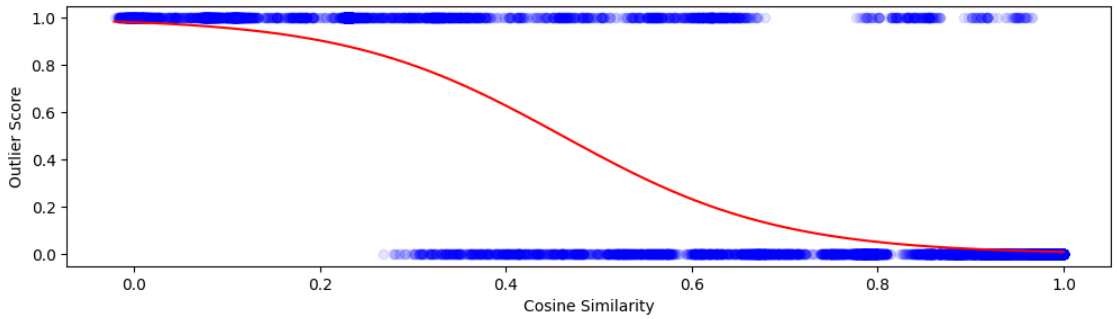
In the example dataset, the cosine similarity was identified as most important feature.
From here, a decision boundary can be derived to distinguish between known and unknown spectra in three different ways:

a) Using the decision boundary from the decision tree: As a first option, we can use the decision tree visualized above.
The optimal decision boundary for the most important feature can be read from the condition of the first split.
In our example, the optimal threshold for the cosine similarity is 0.645.

b) Fitting the decision boundary (logistic regression) Alternatively, the outlier score of a spectrum (1 for outliers and 0 for known spectra) can be plotted against the most important feature.
In our example, the plot shows that known spectra exhibit high cosine similarities while outliers tend to have low cosine similarities.
Next, a sigmoid function is fitted to the data to find the decision boundary. In our case, the optimal threshold is at x0 = 0.47.
You can implement the sigmoid and the fitted parameters in your measurement pipeline to predict the probability of a new spectrum to be an outlier.

[9]:
```
x_data, y_data, x_fit, y_fit =␣
  ↪fit_logistic_regression_for_outlier_feature(df_combined, feature='cos.sim.')

plot_fitted_sigmoid(x_data, y_data, x_fit, y_fit)
```

```
Fitting parameters of sigmoid (x0, k): [ 0.46121577 -8.5881045 ]
```



c) Setting a manual decision boundary Alternatively, the decision boundary for the most important feature can be set manually to separate known and unknown spectra.
On this account, the accuracy, precision and recall for different thresholds between 0 and 1 are calculated and visualized below.
The plot can be read as follows: - **Accuracy** shows how often the outlier detection model is correct overall.

- **Precision** shows how often the outlier detection model is correct when predicting “outlier”.
- **Recall** shows whether the outlier detection model can find all outliers in the data. - A higher threshold for the cosine similarity means that more spectra will be labelled as outliers. - For the extreme threshold of 0, no spectra are labelled as outlier, leading to 90% accuracy as all outlier spectra (10% of the data) are misclassified - Inversely, for an extreme threshold of 1, all spectra are labelled as outliers, leading to 10% accuracy as all known spectra (90% of the data) are misclassified - In our example, a reasonnable choice for the decision boundary would be around 0.5 - 0.7 (balance between accuracy, precision, and recall)

Given this information, you can choose a threshold for outlier detection as a direct criterion and implement it in your measurement pipeline.

[10]:
```
# Initialize lists to store metrics
stepsize = 0.002
thresholds = np.arange(0., 1. + stepsize, stepsize)  # list of thresholds from
 ↪0.5 to 1 in steps of 0.001
accuracies = []
precisions = []
recalls = []

for thresh in thresholds:  # iterate over thresholds

    prediction = SimplePredict(test_cos_sim, cut=thresh)  # distinguish known &
 ↪unknown data at threshold

    # calculate metrics and store in lists
    accuracies.append(accuracy_score(all_targets, prediction))
    precisions.append(precision_score(all_targets, prediction))
    recalls.append(recall_score(all_targets, prediction))

# plot accuracy, precision and recall vs. thresholds
plot_metrics_vs_threshold(thresholds, accuracies, precisions, recalls)
```