eda_microbes

April 26, 2022

1 Microorganism Classification

The rapid and accurate classification of microorganisms is a problem that is of vital importance in the fields of biology and medicine. Medical diagnosis of infections inherently hinges on the identification of bacteria, fungus, or plant microorganism samples taken from a patient. Additionally, microbiologists are constantly tasked with identifying and counting the microorganisms contained in a sample. The microbes dataset provides a dataset covering 10 different bacterial, fungal, and plant microorganism genera along with a number of (potential) characteristic structual features. Microbe classification is a difficult problem due to the similarity between many microbe species, wide ranges of shapes within species, and the undefined orientation of the organism (i.e., how do we account for the physical rotation of the microorganism?). We are given 24 features which are engineered from the pictures which may prove to be useful in microorganism identification. Here are some examples to get an idea: - **EulerNumber** ← The number of objects in the region minus the number of holes in those objects. - **Orientation** ← The overall direction of the shape. The value ranges from -90 degrees to 90 degrees. - **Solidity** ← The ratio of area of an object to the area of a convex hull of the object. Computed as Area/ConvexArea. - **EquivDiameter** ← The diameter of a circle with the same area as the region.

Model Source If you're interested, you can find this work at following github repo: - https://github.com/benjamin-ahlbrecht/microbe-classification

Data Source The data can be found in 2 separate places: Kaggle and Mendeley Data. I'll provide direct links to both here: - https://www.kaggle.com/datasets/sayansh001/microbes-dataset?datasetId=2078183 - https://data.mendeley.com/datasets/f9m85ptmvc/3

1.1 The Microorganisms in Question

We are given 10 different microorganisms to predict. I will list each one here and provide an example picture, so we can get a better idea of what we are dealing with. Notice the distinct shape of each one and some similarities between organism types. Microscopes use light to provide us a representation of an organism and given how light is used, the same organism could be represented in a drastically different way!

1.2 Exploratory Data Analysis

We will explore and analyze the dataset in a number of steps: 1. Data Preparation and Summarization 2. Dimensionality Reduction 3. Model Training 4. Hyperparameter Tuning and Cross Validation 5. Model Testing 6. Results 7. Conclusion

1.2.1 1. Data Preparation and Summarization

[104]:
```
import numpy as np
import pandas as pd
import seaborn as sns

from matplotlib import pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report
from sklearn.model_selection import GridSearchCV
from sklearn import tree

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

[105]:
```
# Load the data in as a dataframe and describe each feature
df = pd.read_csv("Data/microbes.csv")
df.describe()

# It seems that the first feature is simply an index and will not help in our␣
 ↪analysis; drop it\
df.drop(labels=df.columns[0], axis=1, inplace=True)

# Examine our data types so we can prepare them for the model
print(df.dtypes)
print(df["microorganisms"])

# We'll want to encode the various levels of of "microorganisms" feature into␣
 ↪integers, so we can feed them into the model
organisms = df["microorganisms"].unique().tolist()

# Make mappings to and from each organisms
organism_to_int = {organism: i for i, organism in enumerate(organisms)}
```

```
int_to_organism = {val: key for key, val in organism_to_int.items()}

# Apply the encoding
df["microorganisms"] = [organism_to_int[organism] for organism in␣
 ↪df["microorganisms"].tolist()]

df.head()
```

[105]:

	Unnamed: 0	Solidity	Eccentricity	EquivDiameter	Extrema	\
count	30527.00000	30527.000000	30527.000000	30527.000000	30527.000000	
mean	15263.00000	9.677744	19.466921	3.633348	11.871832	
std	8812.53017	4.063437	3.479828	2.210851	6.045135	
min	0.00000	0.000000	0.000000	0.000000	0.000000	
25%	7631.50000	6.570000	17.300000	2.180000	6.790000	
50%	15263.00000	9.350000	20.700000	3.380000	12.000000	
75%	22894.50000	12.600000	22.200000	4.580000	17.200000	
max	30526.00000	23.000000	23.000000	23.000000	23.000000	

	FilledArea	Extent	Orientation	EulerNumber	BoundingBox1	\
count	30527.000000	30527.000000	30527.000000	30527.000000	30527.000000	
mean	0.420022	5.840625	11.751004	22.380901	10.919027	
std	0.875091	3.250999	6.575319	0.962906	6.093280	
min	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.093700	3.280000	6.005000	22.300000	5.690000	
50%	0.229000	5.260000	12.100000	22.600000	10.900000	
75%	0.434500	7.850000	17.200000	22.900000	15.800000	
max	23.000000	23.000000	23.000000	23.000000	23.000000	

	…	ConvexHull3	ConvexHull4	MajorAxisLength	MinorAxisLength	\
count	…	30527.000000	30527.000000	30527.000000	30527.000000	
mean	…	11.046482	11.021988	1.605159	1.014179	
std	…	6.089508	6.089467	1.662537	1.224326	
min	…	0.000000	0.000000	0.000000	0.000000	
25%	…	5.790000	5.755000	0.635000	0.421000	
50%	…	11.000000	11.000000	1.160000	0.745000	
75%	…	15.900000	15.900000	2.070000	1.220000	
max	…	23.000000	23.000000	23.000000	23.000000	

	Perimeter	ConvexArea	Centroid1	Centroid2	Area	\
count	30527.000000	30527.000000	30527.000000	30527.000000	30527.00000	
mean	0.829416	0.254596	11.752783	11.554286	0.80278	
std	1.152165	0.971035	6.029756	5.700637	1.17043	
min	0.000000	0.000000	0.000000	0.000000	0.00000	
25%	0.253000	0.034700	6.570000	7.110000	0.21800	
50%	0.518000	0.085700	12.100000	11.200000	0.51400	
75%	0.968000	0.200000	16.900000	16.200000	0.93400	
max	23.000000	23.000000	23.000000	23.000000	23.00000	

```
              raddi
count  30527.000000
mean       5.214598
std        2.805199
min        0.000000
25%        3.080000
50%        5.320000
75%        7.050000
max       23.000000

[8 rows x 25 columns]

Solidity           float64
Eccentricity       float64
EquivDiameter      float64
Extrema            float64
FilledArea         float64
Extent             float64
Orientation        float64
EulerNumber        float64
BoundingBox1       float64
BoundingBox2       float64
BoundingBox3       float64
BoundingBox4       float64
ConvexHull1        float64
ConvexHull2        float64
ConvexHull3        float64
ConvexHull4        float64
MajorAxisLength    float64
MinorAxisLength    float64
Perimeter          float64
ConvexArea         float64
Centroid1          float64
Centroid2          float64
Area               float64
raddi              float64
microorganisms      object
dtype: object
0        Spirogyra
1        Spirogyra
2        Spirogyra
3        Spirogyra
4        Spirogyra
           …
30522     Ulothrix
30523     Ulothrix
```

```
30524     Ulothrix
30525     Ulothrix
30526     Ulothrix
Name: microorganisms, Length: 30527, dtype: object
```

[105]:

	Solidity	Eccentricity	EquivDiameter	Extrema	FilledArea	Extent	\
0	10.70	15.8	5.43	3.75	0.785	8.14	
1	5.60	18.3	4.14	6.16	0.364	3.51	
2	8.32	19.8	4.63	6.66	0.415	5.85	
3	10.10	17.9	7.29	11.10	1.470	6.30	
4	6.27	20.2	20.10	10.70	14.700	3.97	

	Orientation	EulerNumber	BoundingBox1	BoundingBox2	…	ConvexHull4	\
0	2.15	22.3	2.97	10.90	…	2.97	
1	18.60	22.5	5.41	19.20	…	5.47	
2	21.00	22.4	5.96	10.20	…	5.96	
3	9.94	21.9	8.81	10.70	…	8.88	
4	2.58	11.9	10.20	1.22	…	10.20	

	MajorAxisLength	MinorAxisLength	Perimeter	ConvexArea	Centroid1	\
0	1.34	1.61	0.683	0.195	3.63	
1	1.52	1.52	1.010	0.215	6.01	
2	1.63	1.38	1.110	0.182	6.55	
3	2.04	2.12	0.715	0.371	10.30	
4	7.78	6.21	6.800	4.440	14.00	

	Centroid2	Area	raddi	microorganisms
0	12.10	1.310	7.99	0
1	20.60	0.765	7.99	0
2	11.50	0.953	7.99	0
3	12.00	2.340	7.99	0
4	9.55	17.600	7.99	0

[5 rows x 25 columns]

[115]:
```python
class Data():
    def __init__(self, df, target_col: str, train_proportion: float=0.75, scale:
 ↪ bool=True):
        """
        Class for storing prepared data for the machine-learning pipeline.
        Automatically splits the data into training and testing sets according
        to the train_proportion.

        Parameters
        ----------
        df: Pandas DataFrame
            The prepared data in the form of a Pandas dataframe. This should
```

```
            contain both the features and the target.

        target_col: string
            The name of the taget column in the dataframe; i.e., the feature
            we are trying to classify.

        train_proportion: float, default=0.75
            What proportion of the dataset we should dedicate as training
            data. A complementary amount will be used to test the dataset.

        scale: bool, default=True
            Whether to scale to each feature in the range [0, 1]
        """
        # Extract the labels (y) and the features (x), the feature names and␣
↪the target names
        self.y = df[target_col].to_numpy()
        x_tmp = df.drop(labels=target_col, axis=1)

        self.target = target_col
        self.features_orig = x_tmp.columns.to_list()
        self.features = x_tmp.columns.to_list()

        self.x = x_tmp.to_numpy()
        if scale:
            self.x = MinMaxScaler().fit_transform(self.x)

        # Split our data into training and testing sets
        (self.x_train, self.x_test,
         self.y_train, self.y_test) = train_test_split(
            self.x,
            self.y,
            train_size=train_proportion,
            random_state=11235
        )

        self.train_proportion = train_proportion
        self.test_proportion = 1 - train_proportion

    def __repr__(self):
        return f"""
Data Description:
  Target Variable:  {self.target}
  Feature Size:     {len(self.features_orig)}
  Observation Size: {len(self.y)}
Train/Test Data Description:
  Train Proportion:    {self.train_proportion}
  Test Proportion:     {self.test_proportion}
```

```
  Train Size:          {len(self.y_train)}
  Test Size:           {len(self.y_test)}
  Train Features Size: {self.x_train.shape[1]}
  Test Features Size:  {self.x_test.shape[1]}
"""

    def prune_features(self, mask:list[bool]):
        """
        Remove features from the training and testing sets. The full data
        (x, y) will remain unchanged in case one wishes to re-split the data.

        Arguments
        ---------
        mask: list of bools
            A mask the size of the features dictating whether we should keep a
            feature or remove it.
        """
        self.features = [feature for (feature, keep) in zip(self.features,␣
 ↪mask) if keep]
        self.x_train = self.x_train[:, mask]
        self.x_test = self.x_test[:, mask]
```

[116]:
```
data = Data(df, target_col="microorganisms", train_proportion=0.75)
print(data)
print(data.features)
print(data.x_train)
```

```
Data Description:
  Target Variable:  microorganisms
  Feature Size:     24
  Observation Size: 30527
Train/Test Data Description:
  Train Proportion:    0.75
  Test Proportion:     0.25
  Train Size:          22895
  Test Size:           7632
  Train Features Size: 24
  Test Features Size:  24

['Solidity', 'Eccentricity', 'EquivDiameter', 'Extrema', 'FilledArea', 'Extent',
'Orientation', 'EulerNumber', 'BoundingBox1', 'BoundingBox2', 'BoundingBox3',
'BoundingBox4', 'ConvexHull1', 'ConvexHull2', 'ConvexHull3', 'ConvexHull4',
'MajorAxisLength', 'MinorAxisLength', 'Perimeter', 'ConvexArea', 'Centroid1',
'Centroid2', 'Area', 'raddi']
[[0.89565217 0.56956522 0.05043478 … 0.15695652 0.00281739 0.02404348]
```

```
[0.26434783 0.8826087  0.17608696 … 0.72608696 0.03178261 0.15217391]
[0.43913043 0.94347826 0.24782609 … 0.36521739 0.0626087  0.38913043]
…
[0.48695652 0.65652174 0.08130435 … 0.51304348 0.00704348 0.08043478]
[0.40695652 0.9826087  0.18782609 … 0.46521739 0.03621739 0.15217391]
[0.31347826 0.96086957 0.17217391 … 0.65652174 0.0303913  0.29565217]]
```

1.2.2 Dimensionality Reduction

With 30527 observations, we have a decent amount of data to work with, but we are also working in a high-dimensional space with 24 features. Performing dimensionality reduction may help us prevent from overfitting the model, and a less sparse space will help in classification. Unfortunately, PCA may not be appropriate since one of our features `Orientation` is a circular random variable. Additonally, it is possible that non-linear relationships may exist between the organisms. In this approach, we will attempt to first use a *random forest* to extract the most important features from our *training data* before we begin training the model itself.

[117]:
```
def random_forest_reduction(data, n_estimators: int=100, n_jobs: int=-1,␣
↪verbosity: int=0):
    """
    Uses a Random Forest model to helm select the most informative features
    from a dataset.

    Arguments
    ---------
    data: A Data() object
        The prepared data whose training set will be fed into the model.

    n_estimators: integer, default=100
        How many trees to use in the random forest ensemble

    n_jobs: integer, default=-1
        The number of jobs that can be run in parallel. -1 uses all cores.

    verbosity: integer, default=0
        Whether to output status messages when fitting and predicting the
        model.

    Returns
    -------
    selection_mask: np.ndarray of bools
        An array corresponding to whether a dimension ought to be kept or
        thrown away.
    forest: RandomForestClassifier() object
        The fitted random forest model
    """
    # Instantiate and fit the random forest to the training data
    forest = RandomForestClassifier(
```

```
        n_estimators=n_estimators,
        n_jobs=n_jobs,
        max_depth=5,
        verbose=verbosity
    )
    forest.fit(data.x_train, data.y_train)

    # Create our selector to extract the most informative dimensions
    selector =  SelectFromModel(forest, prefit=True)

    # Extract the selection mask to see what features we ought to keep or toss
    selection_mask = selector.get_support()

    return selection_mask, forest

def visualize_forest(forest, feature_names: list, class_names: list, n_trees:␣
 ↪int=4):
    """
    Visaulize a random forest by specifying a subset of trees as transforming
    them into a figure object.

    Arguments
    ---------
    forest: RandomForestClassifier() object
        The fitted random forest

    feature_names: list
        The corresponding feature names for the random forest

    class_names: list
        The corresponding class names for the random forest

    n_trees: int, default=4
        The number of trees we wish to visualize. To visualize all trees in
        the forest use n_trees=-1

    Returns
    -------
    fig: matplotlib figure
        The matplotlib figure for the random forest visualization
    ax: matplotlib axes
        The corresponding axes for the figure
    """
    assert isinstance(n_trees, int), "Parameter n_trees must be of type int."
    if n_trees == -1:
        n_trees = len(forest.estimators_)
```

```
    class_names = forest.classes_
    trees = forest.estimators_[0:n_trees]

    if n_trees == 1:
        fig, ax = plt.subplots(figsize=(13, 8), tight_layout=True)
        tree.plot_tree(trees[0], filled=True, ax=ax)
        ax.set_title(f"Estimator: 0")
        return fig, ax

    if n_trees <= 3:
        fig, ax = plt.subplots(
            figsize=(21, 13),
            nrows=1,
            ncols=n_trees,
            tight_layout=True
        )

        for i, dt in enumerate(trees):
            tree.plot_tree(dt, filled=True, ax=ax[i])
            ax[i].set_title(f"Estimator: {i}")

        return fig, ax

    # For trees > 3, we will need multiple rows
    nrows = int(np.ceil(n_trees / 3))
    fig, ax = plt.subplots(
        figsize=(21, 13*nrows),
        nrows=nrows,
        ncols=3,
        tight_layout=True
    )

    for i, dt in enumerate(trees):
        col = int(i % 3)
        row = int(np.floor(i / 3))

        tree.plot_tree(dt, filled=True, ax=ax[row, col])
        ax[row, col].set_title(f"Estimator: {i}")

    return fig, ax
```

[118]:
```
# Grab our selection mask and forest model from the random forest reduction
mask, forest = random_forest_reduction(data, n_estimators=150, n_jobs=-1,␣
 ↪verbosity=1)

# Update our data to reflect the dimensionality reduction
data.prune_features(mask)
```

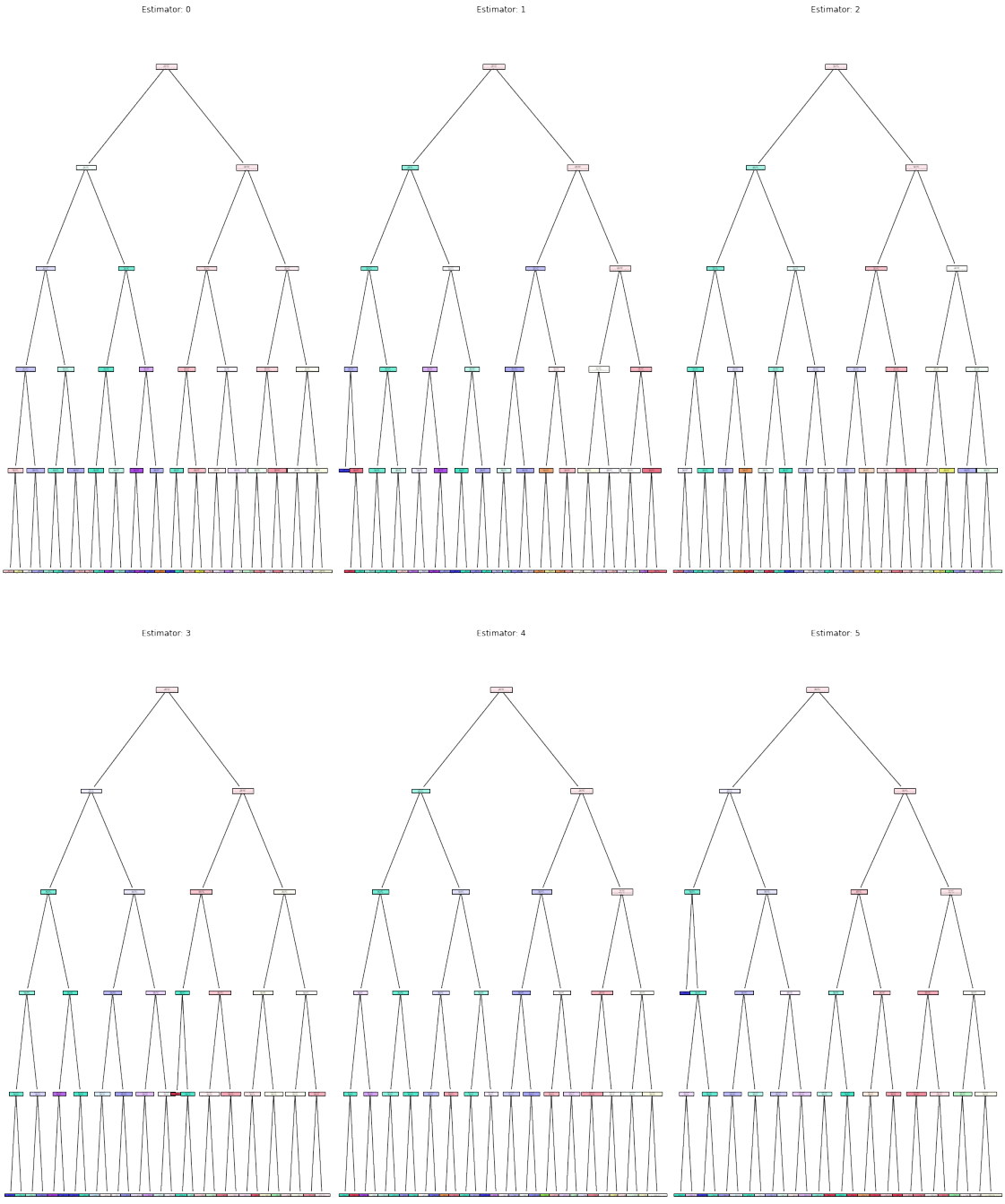
```
print(data)
print(data.features_orig)
print(data.features)

# Visualize some of our decision trees stumps
fig, ax = visualize_forest(forest, organisms, df.columns.to_list(), n_trees=6)
fig.savefig("tree_stumps.pdf")
```

```
[Parallel(n_jobs=-1)]: Using backend ThreadingBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  34 tasks      | elapsed:    0.2s
[Parallel(n_jobs=-1)]: Done 150 out of 150 | elapsed:    0.7s finished


Data Description:
   Target Variable:  microorganisms
   Feature Size:     24
   Observation Size: 30527
Train/Test Data Description:
   Train Proportion:    0.75
   Test Proportion:     0.25
   Train Size:          22895
   Test Size:           7632
   Train Features Size: 12
   Test Features Size:  12

['Solidity', 'Eccentricity', 'EquivDiameter', 'Extrema', 'FilledArea', 'Extent',
'Orientation', 'EulerNumber', 'BoundingBox1', 'BoundingBox2', 'BoundingBox3',
'BoundingBox4', 'ConvexHull1', 'ConvexHull2', 'ConvexHull3', 'ConvexHull4',
'MajorAxisLength', 'MinorAxisLength', 'Perimeter', 'ConvexArea', 'Centroid1',
'Centroid2', 'Area', 'raddi']
['Solidity', 'Eccentricity', 'EquivDiameter', 'FilledArea', 'Extent',
'BoundingBox4', 'MajorAxisLength', 'MinorAxisLength', 'Perimeter', 'ConvexArea',
'Area', 'raddi']
```



1.2.3 Model Training

It appears that we removed a lot of parameters that would likely be highly correlated with one another and not bring much additional information. For example, one would expect the 4 bounding boxes to be related to each-other. The dimensionality-reduction selector likely choose `BoundingBox4` as the most informative bounding box since `BoundingBox1`, `BoundingBox2`, and

`BoundingBox3` are not selected.

Now, let's train our model using a *k-nearest-neighbors* approach. We will utilize the `KNeighborsClassifier()` from *sklearn* as our supervised learning model.

For simplicity, we will examine a simple model here using our **training data**, then tune our hyperparameters and determine the performance of our final model via cross-validation with the testing data.

[110]:
```
knn = KNeighborsClassifier(n_neighbors=4, n_jobs=-1, p=2)
knn.fit(data.x_train, data.y_train)
```

[110]:
```
KNeighborsClassifier(n_jobs=-1, n_neighbors=4)
```

[111]:
```
class DataTesting():
    def __init__(self, labels_pred, labels_truth, decoder: dict=None):
        """
        Class for testing our a fitted model given truth and predicted labels.

        Parameters
        ----------
        labels_pred: np.ndarray
            A numpy array containing the predicted labels from the model
        labels_truth: np.ndarray
            A numpy array containing the true labels from the model
        decoder: dict, default=None
            An optional dictionary used to decode the label values. Has the
            format decoder[encoded_label] -> label_name
        """
        assert isinstance(labels_pred, np.ndarray) and isinstance(labels_truth,␣
↪np.ndarray), "Labels must be a NumPy array."
        assert labels_pred.shape == labels_truth.shape, "Labels must be of␣
↪equal shape."
        self.pred = labels_pred
        self.truth = labels_truth
        self.decoder = {key: val for key, val in decoder.items()}

    def report(self):
        """
        Returns a report on evaluation metrics for the model such as
        specicifity and sensitivity.

        Returns
        -------
        report_df: pd.DataFrame
            The resulting model report in the form of a pandas dataframe
        """
        report = classification_report(
```

```
            self.truth,
            self.pred,
            output_dict=True
        )

        # Make a dataframe and rename the columnns
        report_df = pd.DataFrame.from_dict(report)
        report_df.rename(
            columns={str(key): val for key, val in self.decoder.items()},
            inplace=True
        )

        return report_df

    def confusion_matrix(self):
        """
        Computes the confusion matrix for the model.

        Returns
        -------
        confusion_matrix: pd.DataFrame
            The confusion matrix
        """
        n_features = len(np.unique(self.truth))
        matrix = np.zeros((n_features, n_features))
        for pred, truth in zip(self.pred, self.truth):
            matrix[pred, truth] += 1

        confusion_matrix = pd.DataFrame(matrix)
        confusion_matrix.rename(columns=self.decoder, inplace=True)
        confusion_matrix.rename(index=self.decoder, inplace=True)

        return confusion_matrix

    def confusion_matrix_plot(self, confusion_matrix=None):
        """
        Returns a figure corresponding to the model's confusion matrix.

        Arguments
        ---------
        confusion_matrix: None or pd.DataFrame, default=None
            The confusion matrix to plot against. If None, it will generate
            one from the class attributes
        """
        fig, ax = plt.subplots(figsize=(11, 10), tight_layout=True)
        sns.heatmap(
            data=confusion_matrix,
```

```
            robust=True,
            cmap=sns.dark_palette("#69d", reverse=False, as_cmap=True),
            annot=True,
            fmt=".100g",
            linewidth=0.5,
            ax=ax,
            square=True
        )

        ax.set_xlabel("True Label")
        ax.set_ylabel("Predicted Label")
        ax.set_xticklabels(ax.get_xticklabels(), rotation=45)

        return fig, ax
```

[112]:
```
# We're fitting the TRAINING data just to see our training performance
predictions = knn.predict(data.x_train)
truth = data.y_train

# Create a DataTesting() object for evaluation purposes
data_testing = DataTesting(predictions, truth, decoder=int_to_organism)

# Generate a report
results = data_testing.report()
results

# Create our confusion matrix to visualize where we tend to fail
confusion_matrix = data_testing.confusion_matrix()
confusion_matrix

confusion_matrix_plot, _ = data_testing.confusion_matrix_plot(confusion_matrix)
```

[112]:

	Spirogyra	Volvox	Pithophora	Yeast	Raizopus
precision	0.949749	0.943551	0.918489	0.958469	0.996281
recall	0.816415	1.000000	0.910345	0.991779	1.000000
f1-score	0.878049	0.970956	0.914399	0.974839	0.998137
support	463.000000	3226.000000	1015.000000	2676.000000	1875.000000

	Penicillum	Aspergillus sp	Protozoa	Diatom	Ulothrix
precision	0.958478	0.953464	0.984090	0.953438	0.985425
recall	1.000000	0.930987	1.000000	0.988122	0.935322
f1-score	0.978799	0.942092	0.991981	0.970470	0.959720
support	831.000000	2927.000000	2969.000000	1347.000000	5566.000000

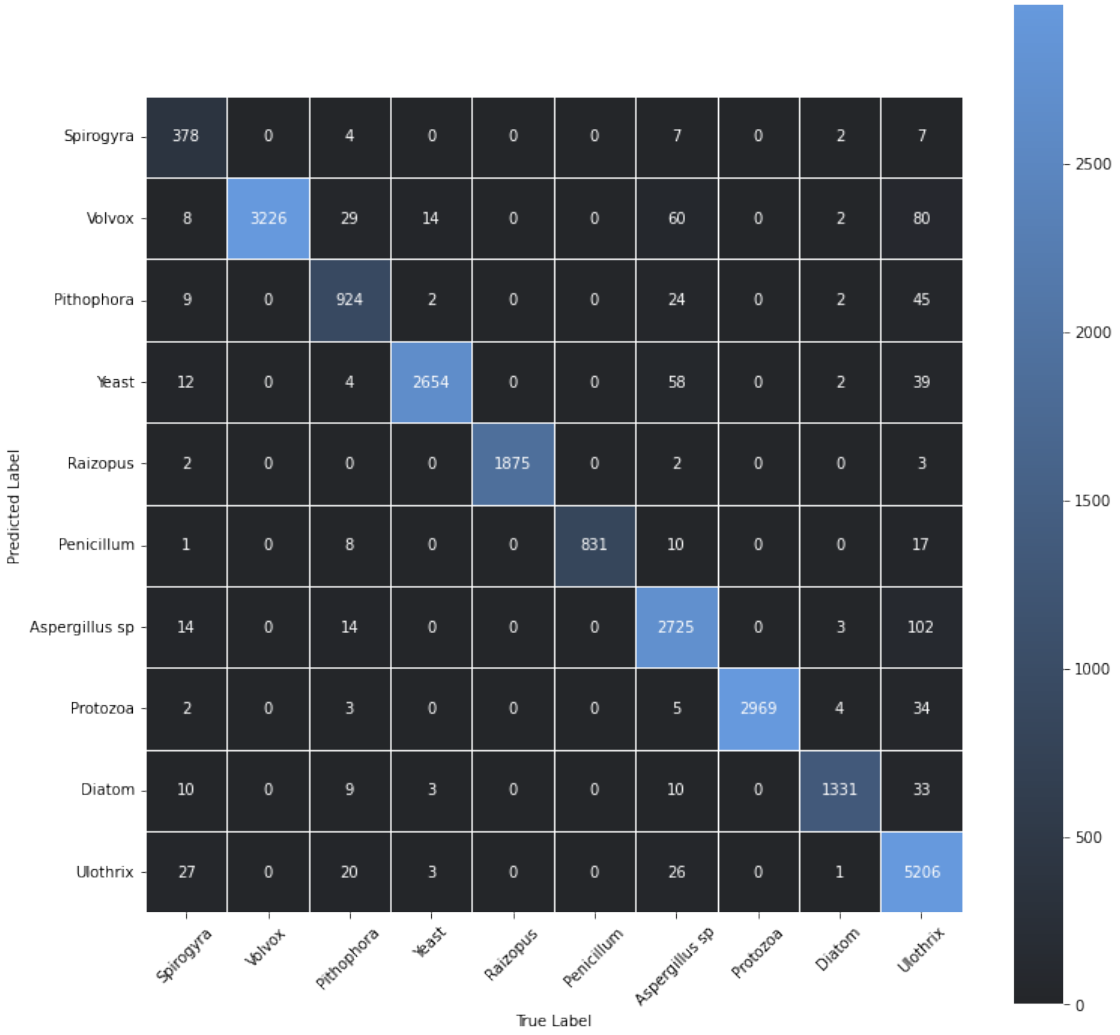
	accuracy	macro avg	weighted avg
precision	0.966106	0.960143	0.966455
recall	0.966106	0.957297	0.966106

f1-score	0.966106	0.957944	0.965810
support	0.966106	22895.000000	22895.000000

[112]:

	Spirogyra	Volvox	Pithophora	Yeast	Raizopus	Penicillum	\
Spirogyra	378.0	0.0	4.0	0.0	0.0	0.0	
Volvox	8.0	3226.0	29.0	14.0	0.0	0.0	
Pithophora	9.0	0.0	924.0	2.0	0.0	0.0	
Yeast	12.0	0.0	4.0	2654.0	0.0	0.0	
Raizopus	2.0	0.0	0.0	0.0	1875.0	0.0	
Penicillum	1.0	0.0	8.0	0.0	0.0	831.0	
Aspergillus sp	14.0	0.0	14.0	0.0	0.0	0.0	
Protozoa	2.0	0.0	3.0	0.0	0.0	0.0	
Diatom	10.0	0.0	9.0	3.0	0.0	0.0	
Ulothrix	27.0	0.0	20.0	3.0	0.0	0.0	

	Aspergillus sp	Protozoa	Diatom	Ulothrix
Spirogyra	7.0	0.0	2.0	7.0
Volvox	60.0	0.0	2.0	80.0
Pithophora	24.0	0.0	2.0	45.0
Yeast	58.0	0.0	2.0	39.0
Raizopus	2.0	0.0	0.0	3.0
Penicillum	10.0	0.0	0.0	17.0
Aspergillus sp	2725.0	0.0	3.0	102.0
Protozoa	5.0	2969.0	4.0	34.0
Diatom	10.0	0.0	1331.0	33.0
Ulothrix	26.0	0.0	1.0	5206.0



1.3 Hyperparameter Tuning and Cross Validation

It seems that our model fits fairly well, but we can go a bit further! Let's add on some hyper-parameter tuning to our model! Since we do not have very many parameters, we can get away using an exhaustive grid search… we will search each combination of parameters and select the one that performs best on the training data to utilize with the testing data. Since we do not want to create a third set of data, a *validation set*, we can use cross-validation to assess the performance of hyper-parameters on the training data.

[113]:
```
# Define our parameter space
parameters = {}
parameters["n_neighbors"] = range(1, 8)
parameters["weights"] = ["uniform", "distance"]
parameters["p"] = [1, 2, float("inf")]
```

```
clf = GridSearchCV(
    estimator=KNeighborsClassifier(),
    param_grid=parameters,
    n_jobs=-1,
    refit=True,
    cv=10,
    verbose=0,
    return_train_score=False
)

clf.fit(data.x_train, data.y_train)

print(clf.best_score_)
print(clf.best_params_)

knn_best = clf.best_estimator_
```

[113]:
```
GridSearchCV(cv=10, estimator=KNeighborsClassifier(), n_jobs=-1,
             param_grid={'n_neighbors': range(1, 8), 'p': [1, 2, inf],
                         'weights': ['uniform', 'distance']})
```

```
0.9804757135416964
{'n_neighbors': 1, 'p': 1, 'weights': 'uniform'}
```

1.3.1 Model Testing

We have finished creating our model. Recall the steps we took in training it: 1. We performed dimensionality reduction to reduce the high-dimensionality space 2. We Assessed the training model with various performance metrics 3. We performed cross validation to pick the best hyper-parameters for the training data Now that wee have finished training the model, we can use the testing data, `data.x_test` and `data.y_test` to test how the model really performs... Does it overfit? underfit? is it accurate? Where does it fail?

[114]:
```
predictions = knn_best.predict(data.x_test)
truth = data.y_test

# Create a DataTesting() object for evaluation purposes
data_testing = DataTesting(predictions, truth, decoder=int_to_organism)

# Generate a report
results = data_testing.report()
results

# Create our confusion matrix to visualize where we tend to fail
confusion_matrix = data_testing.confusion_matrix()
confusion_matrix
```

```
confusion_matrix_plot, _ = data_testing.confusion_matrix_plot(confusion_matrix)
```

[114]:

	Spirogyra	Volvox	Pithophora	Yeast	Raizopus	\
precision	0.949580	0.982047	0.959119	0.981934	1.0	
recall	0.763514	1.000000	0.910448	1.000000	1.0	
f1-score	0.846442	0.990942	0.934150	0.990885	1.0	
support	148.000000	1094.000000	335.000000	924.000000	677.0	

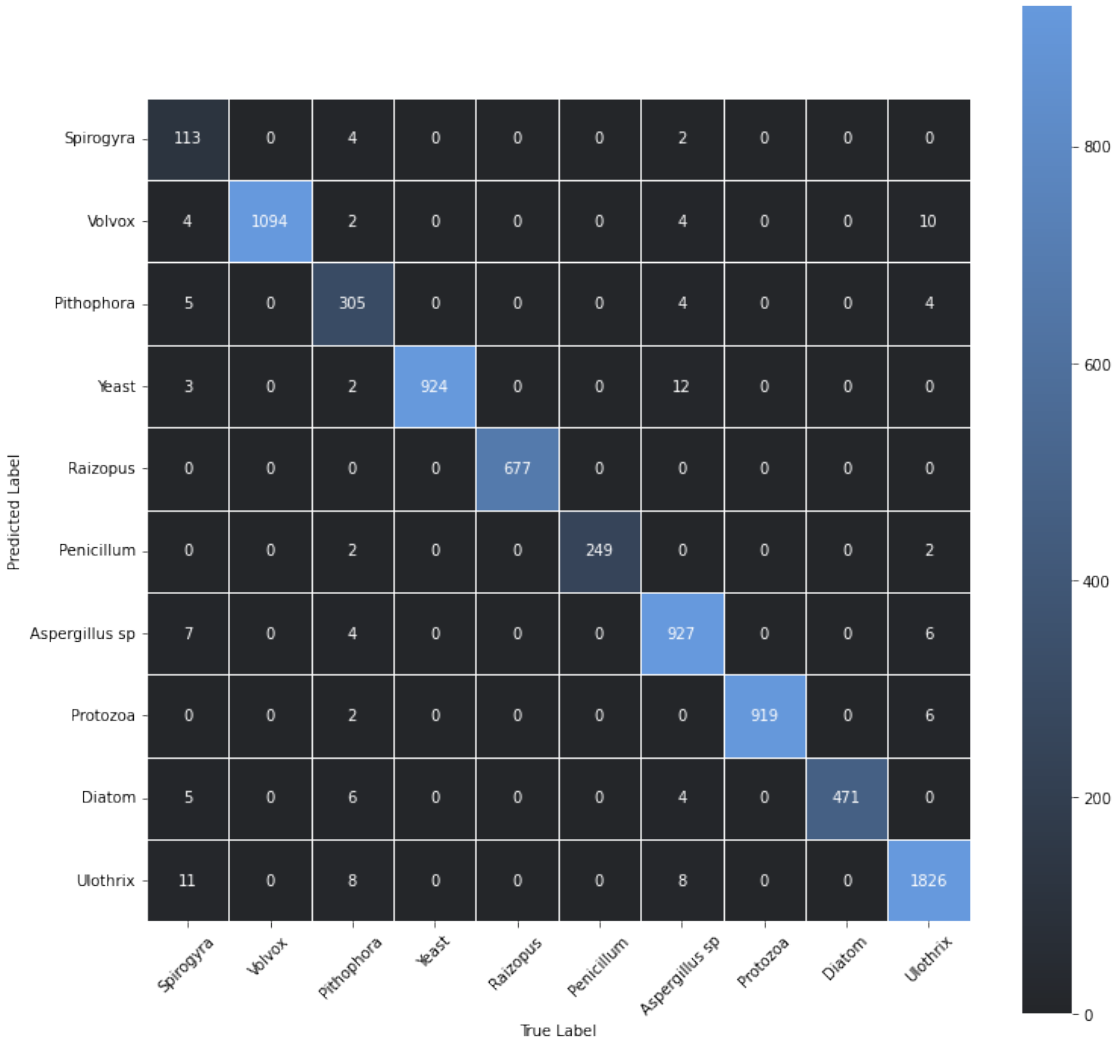
	Penicillum	Aspergillus sp	Protozoa	Diatom	Ulothrix	\
precision	0.984190	0.981992	0.991370	0.969136	0.985429	
recall	1.000000	0.964620	1.000000	1.000000	0.984898	
f1-score	0.992032	0.973228	0.995666	0.984326	0.985163	
support	249.000000	961.000000	919.000000	471.000000	1854.000000	

	accuracy	macro avg	weighted avg
precision	0.98336	0.978480	0.983200
recall	0.98336	0.962348	0.983360
f1-score	0.98336	0.969283	0.983005
support	0.98336	7632.000000	7632.000000

[114]:

	Spirogyra	Volvox	Pithophora	Yeast	Raizopus	Penicillum	\
Spirogyra	113.0	0.0	4.0	0.0	0.0	0.0	
Volvox	4.0	1094.0	2.0	0.0	0.0	0.0	
Pithophora	5.0	0.0	305.0	0.0	0.0	0.0	
Yeast	3.0	0.0	2.0	924.0	0.0	0.0	
Raizopus	0.0	0.0	0.0	0.0	677.0	0.0	
Penicillum	0.0	0.0	2.0	0.0	0.0	249.0	
Aspergillus sp	7.0	0.0	4.0	0.0	0.0	0.0	
Protozoa	0.0	0.0	2.0	0.0	0.0	0.0	
Diatom	5.0	0.0	6.0	0.0	0.0	0.0	
Ulothrix	11.0	0.0	8.0	0.0	0.0	0.0	

	Aspergillus sp	Protozoa	Diatom	Ulothrix
Spirogyra	2.0	0.0	0.0	0.0
Volvox	4.0	0.0	0.0	10.0
Pithophora	4.0	0.0	0.0	4.0
Yeast	12.0	0.0	0.0	0.0
Raizopus	0.0	0.0	0.0	0.0
Penicillum	0.0	0.0	0.0	2.0
Aspergillus sp	927.0	0.0	0.0	6.0
Protozoa	0.0	919.0	0.0	6.0
Diatom	4.0	0.0	471.0	0.0
Ulothrix	8.0	0.0	0.0	1826.0



1.3.2 Results

The final model performed very well on the testing data with approximately 98.3% accuracy overall! It appears that `Raizopus` is the easiest organism to classify with 100% accuracy. - The most extreme performance metric is the `recall` metric for `Spirogyra`: 76.4%. This means that out of the 148 `Spirogyra` total only 113 were correctly identified to be `Spirogyra` by the model. In the confusion matrix, this can be calculated by examining the number of correctly categorized `Spirogyra` in the diagonal of the confusion matrix and dividing it by the sum of the `Spirogyra` column. The precision for `Spirogyra` is calculated similarly, but instead of summing the `Spirogyra` column, we sum the `Spirogyra` row. - It appears that the most the model most frequently mis-characterizes an `Aspergillus sp` as a `Yeast` (12 wrong predictions), a `Spirogyra` as an`Ulothrix` (11 wrong predictions), and a `Ulothrix` as a `Volvox` (10 wrong predictions)

Prediction	Truth	Count
		12
		11
		10

1.3.3 Conclusion

Overall, it appears that the model works very well. However, this does not mean that no improvements can be made. Notice how feature selection and hyper-parameter optimization were entirely separate. However, there is no reason why feature selection has no effect on the optimal hyper-parameter values. Because of this, it may be possible to reach better model performance by performing feature selection and hyper-parameter optimization at the same time.

Additionally, the high performance is very likely thanks to the genius feature-engineering! If we simply used the straight pixels from the images, we would very likely have a much harder time classifying the microorganisms. Instead, our engineered features explained far greater variance than individuals pixels could. This also helped us work in a much lower-dimensional space, aptly avoiding the curse of dimensionality.