project1-obesityclassification

October 7, 2024

0.1 Obesity Dataset Classification Exploration

This project attempts to explore and understand Machine Learning classification techniques using a multiclass dataset, sourced from kaggle. A number of machine learning concepts are discussed, and a variety of classification models are used and analyzed.

0.1.1 What Are We Trying to Learn?

The goal of the project will be to maximize the performance of a variety of machine learning classifiers from sklearn for our multiclass dataset. These are: decision trees, random forests, gradient boosting, svm, and knn. The project will also serve to learn the hyperparameter tuning tool GridsearchCV, feature selection, and evaluation metrics or model scoring.

0.1.2 Why Does It Matter?

This project is for the CU Boulder CSPB machine learning course. While regular course material is both relevent and plentiful, the realm of machine learning techniques and tools is incredibly vast and cannot be covered only by reading and standard homework material. An opportunity to explore is invaluable, because it involves attempting to answer unexpected questions and grows experience in a way which is more personable.

0.1.3 Data Link and Info

The Sinop University Obesity Dataset was collected as a research project by Nigmet Koklu and Süleyman Alpaslan Sulak. The authors conducted an online survey in order to collect the observations available in the dataset with the intention of analyzing it through machine learning techniques. They have provided a single table in a Microsoft Excel format. The table is comprised of 1610 of the responses to the survey, with 15 total columns including the response labels. The provided survey responses are comprehensive; no values are missing. Nearly every feature is categorical, and coded numerically. 'Table Attributes' provides a mapping of the encoded values as well as respecive counts in parentheses. The only properly numeric features are Age and Height. They are stored as integers, coded as years and centimeters, respectively.

Table Attributes

Sex	Overweight/Obese Family	Consumption of Fast Food	Frequency of Consuming Vegetables	Number of Main Meals Daily
1) Male (712)	1) Yes (266)	1) Yes (436)	1) Rarely (400)	1) One to Two (444)
2) Female (898)	2) No (1344)	2) No (1174)	2) Sometimes (708)	2) Three (928)
			3) Always (502)	3) Three-Plus (238)

Age	Height	Smoking	Liquid Intake Daily	Calculates Calories
In Years	**In Centimeters**	1) Yes (492)	1) Less Than a Liter (456)	1) Yes (286)
		2) No (1118)	2) One to Two liters (523)	2) No (1324)
			3) More Than Two (631)	

Food Intake Between Meals	Physical Exercise	Hours Tech Use per Day	Transportation Used	Class
1) Rarely (346)	1) None (206)	1) Zero to Two (382)	1) Automobile (660)	1) Underweight (71)
2) Sometimes (564)	2) One to Two Days (290)	2) Three to Five (826)	2) Motorbike (94)	2) Normal (658)
3) Usually (417)	3) Three to Four (370)	3) More than Five (402)	3) Bike (116)	3) Overweight (592)
4) Always (283)	4) Five to Six Days (358)		4) Public Transit (602)	4) Obese (287)
	5) Six-Plus Days (386)		5) Walking (138)	

0.1.4 Preliminary Considerations

A convenience of the preemptively clean nature of the dataset is that it can make many wrangling techniques unnecessary. Like other kaggle datasets this one is the result of other research, making it more ready for ingestion into a model. Although, ideally this dataset would be larger. Because of this and some unevenness in classes, a test split of twenty-five percent is used. Attributes included here are largely categorical, and already numerically encoded. This would allow the nonparametric models used for the project to be built immediately, were we not concerned with attempting to further optimize features prior to parameter tuning. Also of benefit, categorical attributes principally lack outliers or unusable values. The total number of features involved are

easily handled for these model types, but we will be attempting to alter what features we actually use before finalizing the analysis. We have already seen a table of the feature encodings and sizes, but before we can alter the feature set, we need to examine their associations.

0.1.5 Imports, Basic Info, and Some Example Observations

[33]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import copy
import scipy as sp
import scipy.stats as stats
import statsmodels.formula.api as smf
import statsmodels.api as sm
from sklearn.metrics import accuracy_score
from sklearn.metrics import balanced_accuracy_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import RocCurveDisplay
from sklearn.preprocessing import LabelBinarizer
from sklearn.multiclass import OneVsRestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.base import clone
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve


ob = pd.read_excel("data/Obesity_Dataset.xlsx")
classifier_names = ['DecisionTreeClassifier', 'SVC', 'RandomForestClassifier',␣
 ↪'GradientBoostingClassifier', 'KNeighborsClassifier']
classifiers = [DecisionTreeClassifier, SVC, RandomForestClassifier,␣
 ↪GradientBoostingClassifier, KNeighborsClassifier]
models_dict = {}
RANDOMSTATE = 42

param_grids = {
    'DecisionTreeClassifier': {'max_depth': [i for i in range(1,15)],
```

```
                                'ccp_alpha': [0, .1, .2, .5, 1],
                                'class_weight': [None, 'balanced'],
                                'criterion': ['gini', 'entropy', 'log_loss'],
                                'random_state': [RANDOMSTATE]},

    'SVC': {'C': [i for i in range(90,101)],
            'gamma': ['auto', 'scale', .01],
            'kernel': ['rbf', 'linear', 'sigmoid'],
            'random_state': [RANDOMSTATE]},

    'RandomForestClassifier': {'n_estimators': [i*10 for i in range(5,13)],
                               'max_depth': [i for i in range(1,15)],
                               #'ccp_alpha': [0, .1, 1],
                               'criterion': ['gini', 'entropy', 'log_loss'],
                               'max_features': ['sqrt', 'log2', None],
                               'random_state': [RANDOMSTATE]},

    'GradientBoostingClassifier': {'n_estimators': [i*10 for i in range(1,15)],
                                   'max_depth': [i for i in range(1,10)],
                                   #'ccp_alpha': [0, .1, 1],
                                   'learning_rate': [.001, .01, .1, .4],
                                   'random_state': [RANDOMSTATE]},

    'KNeighborsClassifier': {'n_neighbors': [i for i in range(1,20)],
                             'algorithm': ['ball_tree', 'kd_tree', 'brute',␣
 ↪'auto'],
                             'weights': ['distance', 'uniform'],
                             'p': [1, 2]}
}

def ParamTournament(x_tr, x_te, y_tr, y_te, params_list,␣
 ↪scoring_param='accuracy', clf=DecisionTreeClassifier):
    top = 0, {}
    for params in params_list:
        m = clf(**params).fit(x_tr, y_tr)
        y_hat = m.predict(x_te)

        if scoring_param == 'accuracy':
            test_score = accuracy_score(y_true=y_te, y_pred=y_hat)

        elif scoring_param == 'balanced_accuracy':
            test_score = balanced_accuracy_score(y_true=y_te, y_pred=y_hat)

        elif scoring_param == 'f1_weighted':
            test_score = f1_score(y_true=y_te, y_pred=y_hat, average='weighted')

        top = (test_score, params) if test_score > top[0] else (top[0], top[1])
```

```
    return top[1]


def pipe_scores(clf, p_grid, x_tr, x_te, y_tr, y_te, scoring_param='accuracy'):
    '''pipeline for a given format of classifier'''

    models = GridSearchCV(estimator=clf(), param_grid=p_grid,␣
 ↪scoring=scoring_param, n_jobs=-1, cv=4)

    models.fit(x_tr, y_tr)

    pl = pd.DataFrame(models.cv_results_).sort_values('rank_test_score').
 ↪head(10)['params']
    params = ParamTournament(x_tr, x_te, y_tr, y_te, pl, scoring_param, clf)

    idx = models.cv_results_['params'].index(params)
    cv_score = models.cv_results_['mean_test_score'][idx]

    m0 = clf(**params).fit(x_tr, y_tr)

    y_hat = m0.predict(x_te)

    if scoring_param == 'accuracy':
        test_score = accuracy_score(y_true=y_te, y_pred=y_hat)
        train_score = accuracy_score(y_true=y_tr, y_pred=m0.predict(x_tr))

    elif scoring_param == 'balanced_accuracy':
        test_score = balanced_accuracy_score(y_true=y_te, y_pred=y_hat)
        train_score = balanced_accuracy_score(y_true=y_tr, y_pred=m0.
 ↪predict(x_tr))

    elif scoring_param == 'f1_weighted':
        test_score = f1_score(y_true=y_te, y_pred=y_hat, average='weighted')
        train_score = f1_score(y_true=y_tr, y_pred=m0.predict(x_tr),␣
 ↪average='weighted')

    return [cv_score, train_score, test_score]



def pipeline(clf, p_grid, x_tr, x_te, y_tr, y_te, scoring_param='accuracy',␣
 ↪m_d=models_dict):
    '''pipeline for a given format of classifier'''

    models = GridSearchCV(estimator=clf(), param_grid=p_grid,␣
 ↪scoring=scoring_param, n_jobs=-1, cv=4)
```

```
    models.fit(x_tr, y_tr)

    pl = pd.DataFrame(models.cv_results_).sort_values('rank_test_score').
 ↪head(10)['params']
    params = ParamTournament(x_tr, x_te, y_tr, y_te, pl, scoring_param, clf)

    idx = models.cv_results_['params'].index(params)
    cv_score = models.cv_results_['mean_test_score'][idx]

    m0 = clf(**params).fit(x_tr, y_tr)
    m_d[f'{clf.__name__}'] = { 'params': m0.get_params(), 'classifier': clf }

    y_hat = m0.predict(x_te)
    cm = confusion_matrix(y_true=y_te, y_pred=y_hat)

    if scoring_param == 'accuracy':
        test_score = m0.score(y_true=y_te, y_pred=y_hat)
        train_score = m0.score(y_true=y_tr, y_pred=m0.predict(x_tr))

    elif scoring_param == 'balanced_accuracy':
        test_score = balanced_accuracy_score(y_true=y_te, y_pred=y_hat)
        train_score = balanced_accuracy_score(y_true=y_tr, y_pred=m0.
 ↪predict(x_tr))

    elif scoring_param == 'f1_weighted':
        test_score = f1_score(y_true=y_te, y_pred=y_hat, average='weighted')
        train_score = f1_score(y_true=y_tr, y_pred=m0.predict(x_tr),␣
 ↪average='weighted')

    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt="d", cmap="rocket_r")

    plt.xlabel("Predicted Label")
    plt.ylabel("True Label")
    plt.title("Confusion Matrix")

    print(m0.get_params())
    print(f"{clf.__name__} Mean Cross-Validation score: ", cv_score)
    print(f"{clf.__name__} Training Score: ", train_score)
    print(f"{clf.__name__} Testing Score: ", test_score)

    return models.cv_results_


ob.info()
ob.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1610 entries, 0 to 1609
Data columns (total 15 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   Sex                                   1610 non-null   int64
 1   Age                                   1610 non-null   int64
 2   Height                                1610 non-null   int64
 3   Overweight_Obese_Family               1610 non-null   int64
 4   Consumption_of_Fast_Food              1610 non-null   int64
 5   Frequency_of_Consuming_Vegetables     1610 non-null   int64
 6   Number_of_Main_Meals_Daily            1610 non-null   int64
 7   Food_Intake_Between_Meals             1610 non-null   int64
 8   Smoking                               1610 non-null   int64
 9   Liquid_Intake_Daily                   1610 non-null   int64
 10  Calculation_of_Calorie_Intake         1610 non-null   int64
 11  Physical_Excercise                    1610 non-null   int64
 12  Schedule_Dedicated_to_Technology      1610 non-null   int64
 13  Type_of_Transportation_Used           1610 non-null   int64
 14  Class                                 1610 non-null   int64
dtypes: int64(15)
memory usage: 188.8 KB
```

[33]:

	Sex	Age	Height	Overweight_Obese_Family	Consumption_of_Fast_Food	Frequency_of_Consuming_Vegetables	Number_of_Main_Meals_Daily	Food_Intake_Between_Meals	Smoking	Liquid_Intake_Daily	Calculation_of_Calorie_Intake	Physical_Excercise
0	2	18	155	2	2	3	1	3	2	1	2	3
1	2	18	158	2	2	3	1	1	2	1	2	1
2	2	18	159	2	2	2	1	3	2	3		
3	2	18	162	2	2	2	2	2	2	2		
4	2	18	165	2	1	2	1	3	2	1		

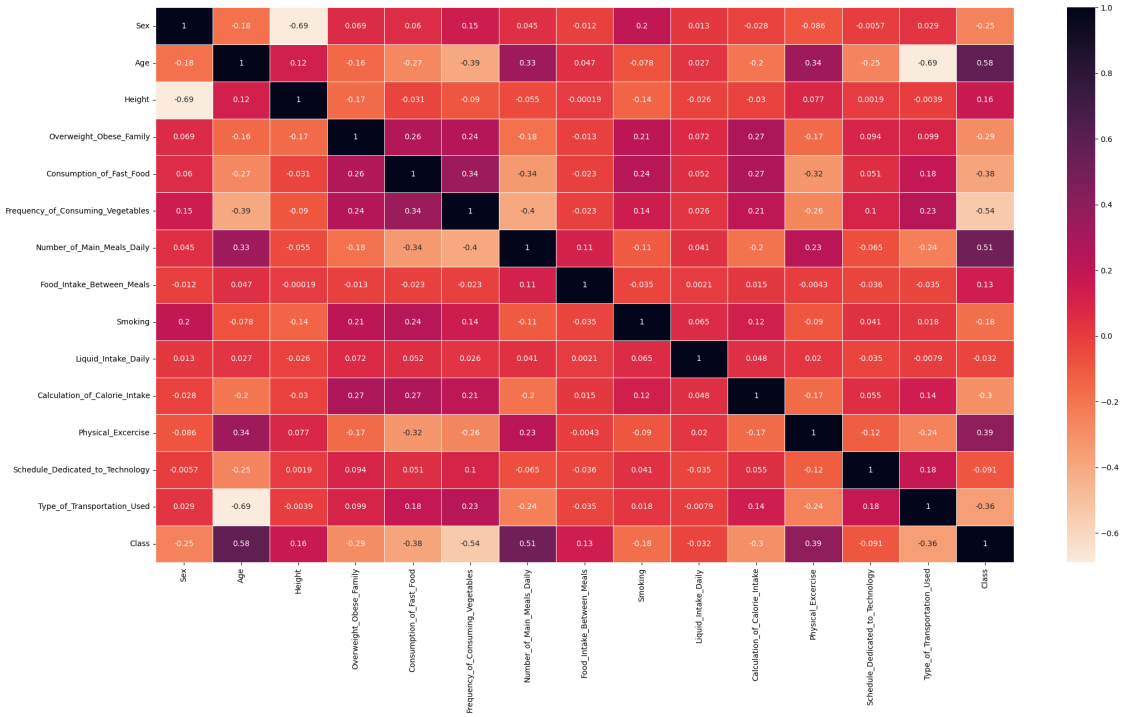
```
2                                2                                2
3                                2                                1
4                                2                                3

   Schedule_Dedicated_to_Technology  Type_of_Transportation_Used  Class
0                                 3                            4      2
1                                 3                            3      2
2                                 3                            4      2
3                                 3                            4      2
4                                 3                            2      2
```

0.1.6 Feature Associations

The correlation matrix and pairplot are our first stop. We need to consider whether predictors suggest colinearity, or show no association with the response. Looking to the pairplot, ease of interpretability does not seem to be helped by the fact that so many of the available features are categorical. Distributions between predictors are ideal when they show random distribution on the pairplot, but having few category options (low cardinality) occludes the distribution in most cases. Numeric categories Age and Height do distribute evenly.

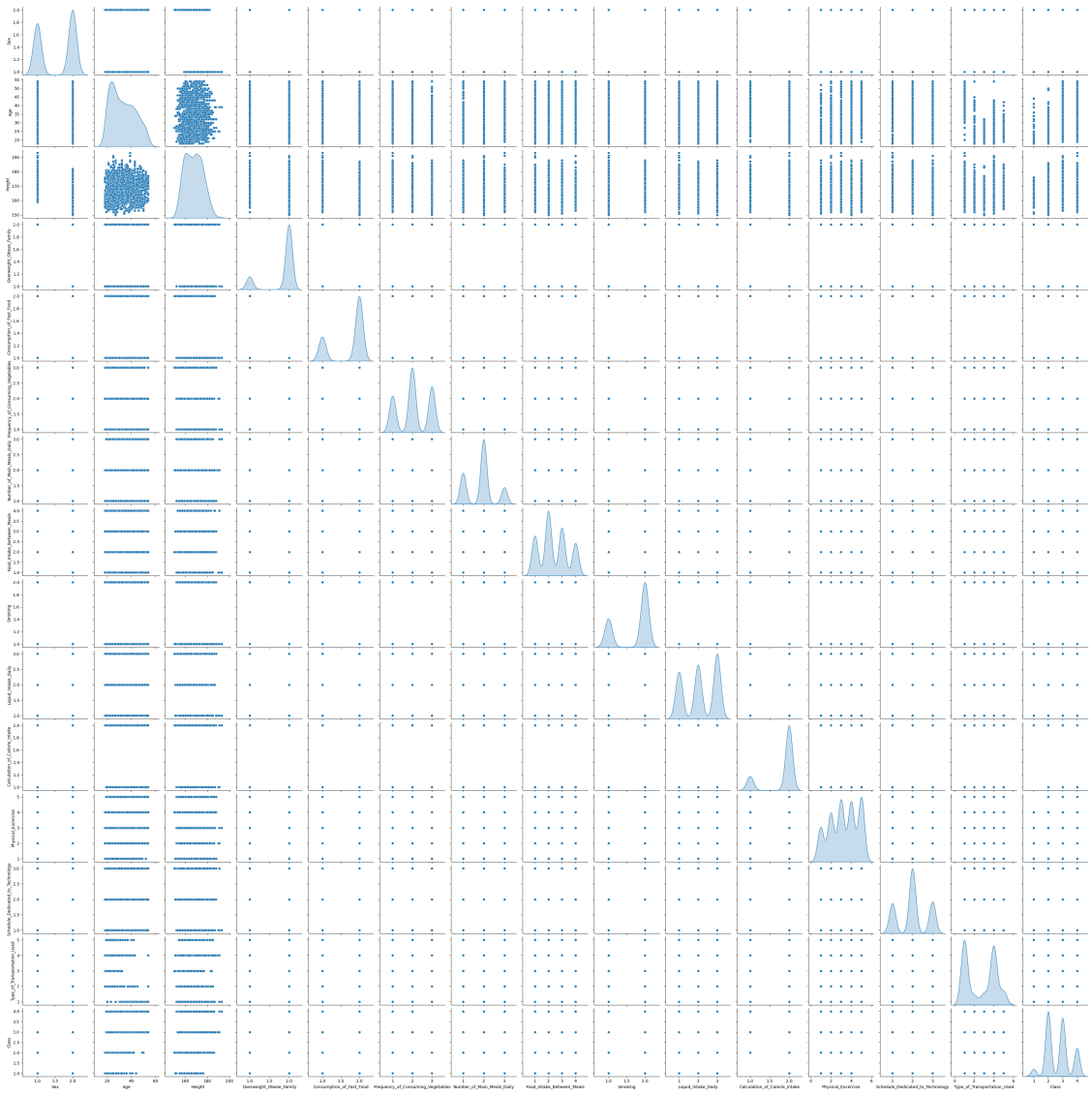
[34]:
```
plt.figure(figsize=(25, 13))
sns.heatmap(ob.corr(), annot=True, cmap='rocket_r', linewidth=.5)
```

[34]: <Axes: >



[35]:
```
sns.pairplot(ob, diag_kind='kde')
```

[35]: <seaborn.axisgrid.PairGrid at 0x7ff108c21ed0>



[36]:
```
y=ob['Class']
X=ob.drop(labels='Class', axis=1)

x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.25,␣
 ↪random_state=RANDOMSTATE)
```

0.1.7 A Way Into Feature Elimination

sklearn page used to learn how to visualize feature importances

Non-parametric models (k-Nearest Neighbors, Decision Trees, Random Forests, and Support Vector Machines where the kernel is non-linear) do not assume any fixed form for the function they model from the data. They rely on the actual data distribution to make predictions. Even though non-parametric models do not have fixed parameters like weights in linear models, feature importance metrics - especially from models like Random Forests - can help to identify features that contribute most to prediction.

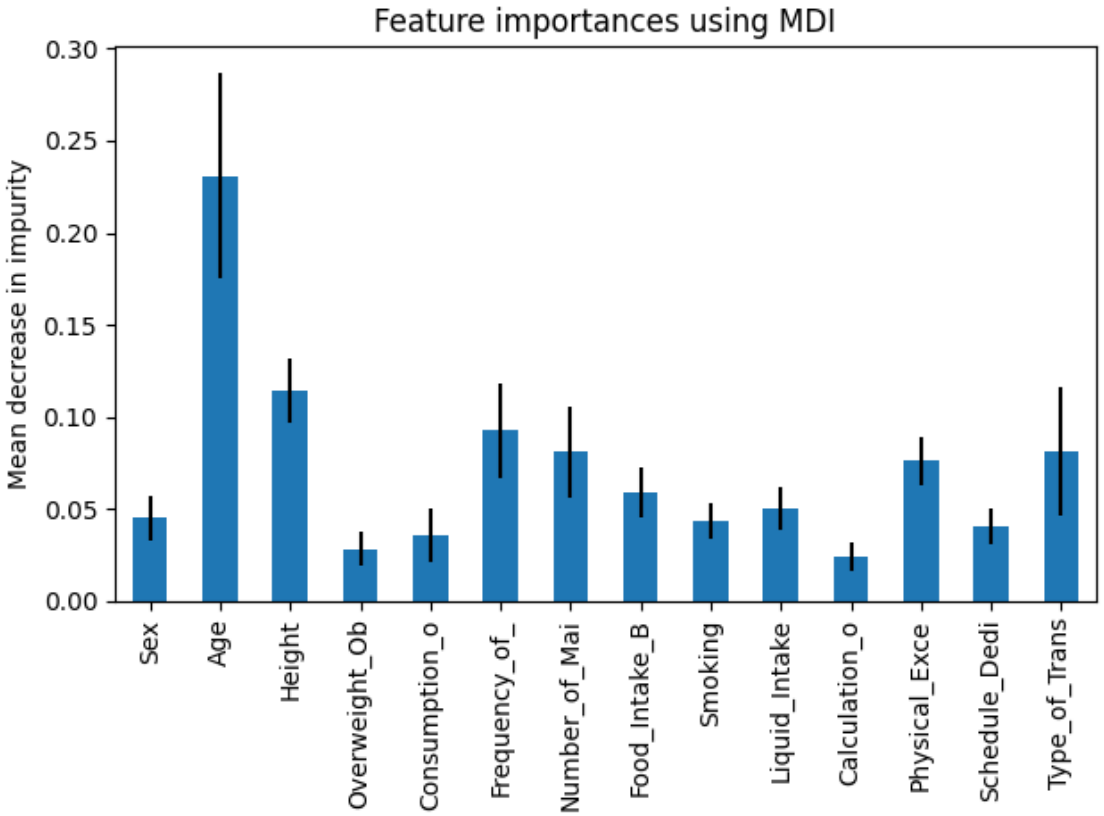
In lieu of finding an interaction term while initially exploring the data I have decided to attempt to improve model performance by simplifying the feature set. Where this is possible this would mean that some feature or features are sufficiently noisy as to be better off excluding them. This is an important goal for the project, because doing so would help to reduce overfitting of our models. The question then is where to start. Some method is needed to prioritize features available and potentially come up with a route by which we might simplify.

Since several of our selected classifiers are tree based, it makes sense to use MDI to measure importance and go from there. Tree methods (which we have several of) make splits in the data in a way which prioritizes as "pure" of a result to one side as possible. Features which decrease impurity significantly during splits are considered "important". The RandomForestClassifier can directly provide an importance valuation, and we can use lack of importance to consider what to try and exclude first.

[37]:
```
forest = RandomForestClassifier().fit(x_train, y_train)
abbreviations = [x[:13] for x in x_train.columns]
importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)

importances = pd.Series(forest.feature_importances_, index=x_train.columns)

fig, ax = plt.subplots()
pd.Series(forest.feature_importances_, index=abbreviations).plot.bar(yerr=std,␣
 ↪ax=ax)
ax.set_title("Feature importances using MDI")
ax.set_ylabel("Mean decrease in impurity")
fig.tight_layout()
print(importances.sort_values(ascending=False))
```

```
Age                                 0.230919
Height                              0.114028
Frequency_of_Consuming_Vegetables   0.092394
Type_of_Transportation_Used         0.081073
Number_of_Main_Meals_Daily          0.080987
Physical_Excercise                  0.075852
Food_Intake_Between_Meals           0.058808
Liquid_Intake_Daily                 0.049826
Sex                                 0.044762
Smoking                             0.043509
Schedule_Dedicated_to_Technology    0.040316
Consumption_of_Fast_Food            0.035890
Overweight_Obese_Family             0.027863
```

```
Calculation_of_Calorie_Intake        0.023773
dtype: float64
```



The chart above shows that Age is the most conducive to a reduction in impurity and implies it to be the most predictive feature available for weight classification, which seems reasonable. The lowest in MDI are Overweight_Obese_Family and Calculation_of_Calorie_Intake. Both of these features are simple binary choices, and apparently are not currently able to provide much improvement for purity under the current feature set. It is possible that these two introduce some noise to the data, as there is no guarantee that a person who responds that they count calories cannot also be overweight or obese, and the proportion to whether a respondent provides a 'yes' on overweight or obese family seems one sided (considering counts of responses in the table above).

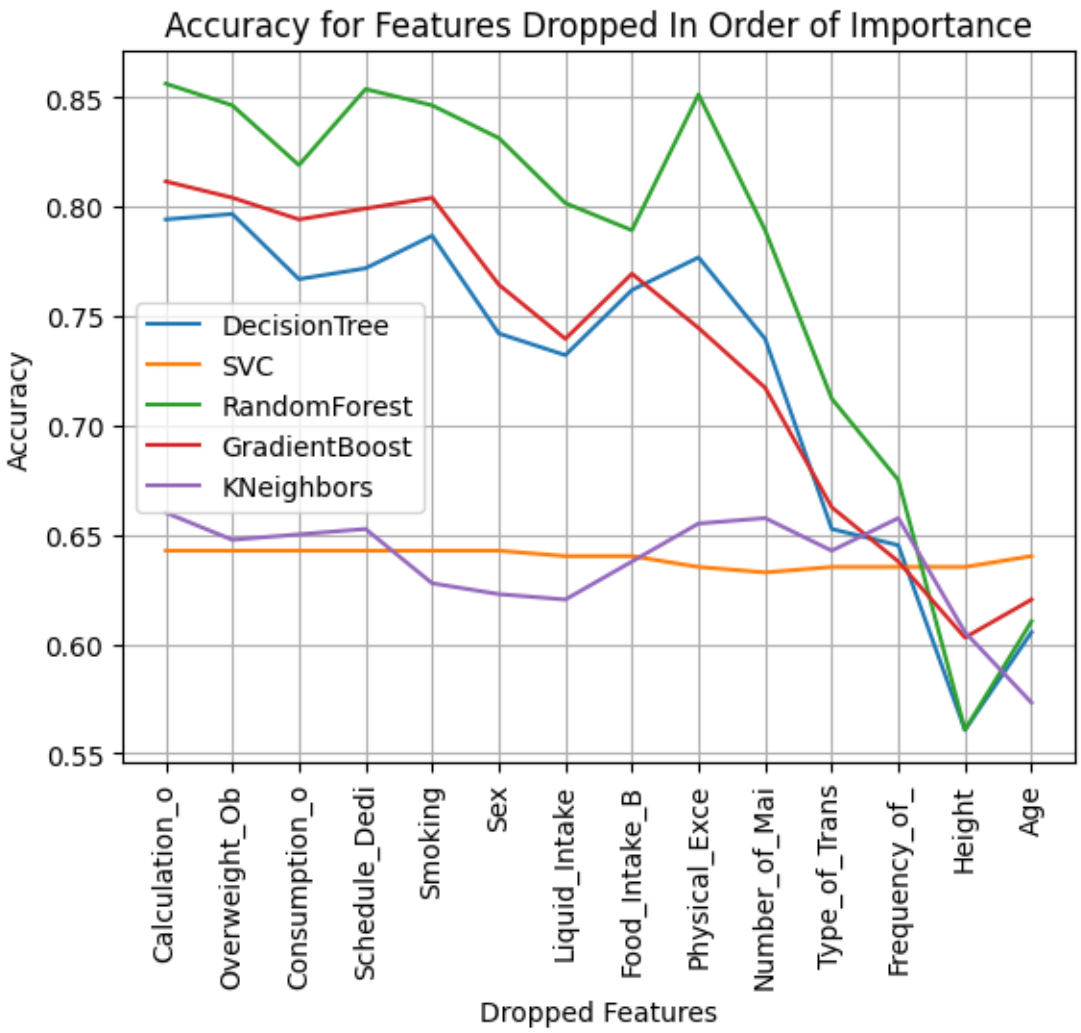
Moving further, we need to make some assumptions. We need a way to iteratively remove features and interpret the results. Because Age is ranked highest, we can use it as a final remaining feature in that iterative process and compare from there.

[38]:
```
to_reduce = list(importances.sort_values(ascending=True).index)

dt_d_redux, svm_d_redux, rf_d_redux, gb_d_redux, knn_d_redux = [], [], [], [],␣
↪[]

for i in range(len(to_reduce)):
```

```
    x_tr = x_train.drop(labels=to_reduce[:i], axis=1)
    x_te = x_test.drop(labels=to_reduce[:i], axis=1)

    dt_d_redux.append(DecisionTreeClassifier(random_state=RANDOMSTATE).
 ↪fit(x_tr, y_train).score(x_te, y_test))
    svm_d_redux.append(SVC(random_state=RANDOMSTATE).fit(x_tr, y_train).
 ↪score(x_te, y_test))
    rf_d_redux.append(RandomForestClassifier(random_state=RANDOMSTATE).
 ↪fit(x_tr, y_train).score(x_te, y_test))
    gb_d_redux.append(GradientBoostingClassifier(random_state=RANDOMSTATE).
 ↪fit(x_tr, y_train).score(x_te, y_test))
    knn_d_redux.append(KNeighborsClassifier().fit(x_tr, y_train).score(x_te,␣
 ↪y_test))
```

[39]:
```
drops = np.argmax(np.array([dt_d_redux, svm_d_redux, rf_d_redux, gb_d_redux,␣
 ↪knn_d_redux]), axis=1)
print("Number of drops for best score: ", drops)
x = [x[:13] for x in to_reduce]

fig, ax = plt.subplots()
ax.set_xlabel("Dropped Features")
ax.set_ylabel("Accuracy")
ax.set_title("Accuracy for Features Dropped In Order of Importance")
ax.plot(x, dt_d_redux, label="DecisionTree")
ax.plot(x, svm_d_redux, label="SVC")
ax.plot(x, rf_d_redux, label="RandomForest")
ax.plot(x, gb_d_redux, label="GradientBoost")
ax.plot(x, knn_d_redux, label="KNeighbors")
plt.xticks(rotation=90)
plt.grid(True)
ax.legend()
plt.show()
```

```
Number of drops for best score:  [1 0 0 0 0]
```



0.1.8 Latent Patterns

Consider the above line chart. We can observe behavior in the trend lines which measure each classifier's accuracy as features are successively eliminated. The models used here all have default parameter values, which is convenient to implement, but it is important to remember that the end goal of any feature exclusion we might make is intended to result in a more performant model once tuning is performed.

We can see from the chart that performances of the models (especially tree based ones) experience multiple rebounds at different points of sequential elimination. These rebounds coincide with the elimination of Consumption_of_Fast_Food, Schedule_Dedicated_to_Technology, Liquid_intake_Daily, Food_Intake_Between_Meals, and Height (mostly). Because there is not a direct reduction in accuracy when they are dropped, this could mean our models would be better off without them. However, without benchmarking performance after hyperparamter tuning, there will not be much assurance as to whether these trends are indications of real predictive improvement or not. Further confounding interpretation here is the fact that when we con-

sult the correlation matrix above with respect to, for example, Consumption_of_Fast_Food and Food_Intake_Between_Meals we can see that Fast Food regularly has some correlation (positive or negative) more extreme than 0.1 or 0.2 in several cases for other predictors, and yet Between Meals shows very consistent low correlation across predictors. It may be that eliminating these two specific candidates would be most useful to eliminate noise within the training data, or that by the time so many features of lower importance are eliminated that we observe a performance uplift due to a sheer volume of noise being removed collectively. Only a gauntlet of hyperparameter tuning for all of these possible eliminations (and theoretically every possible combination of features) will provide greater certainty.

The following DataFrame and line chart were constructed by using the 'pipe_scores' function provided above. This function was used to perform hyperparameter tuning for each of our classifiers, specific to each feature subset. Consequently, this involved several tuning loops which would collectively run for around an hour in order to complete. So, the resulting score data is compiled into a csv file in order to save time in evaluating the results. Column names are tagged to indicate what subset of features get excluded.

"Full" : all fourteen features kept

"FF" : only Consumption_of_Fast_Food eliminated

"DT" : only Schedule_Dedicated_to_Technology eliminated

"BM" : only Food_Intake_Between_Meals eliminated

"LI" : only Liquid_Intake_Daily eliminated

"H" : only Height eliminated

"PBA" : Consumption_of_Fast_Food, Liquid_Intake_Daily, Food_Intake_Between_Meals eliminated, and Height eliminated

"KS" : lowest eight (by importance) features eliminated, ie Calculation_of_Calorie_Intake through Food_Intake_Between_Meals

[40]:
```
df3 = pd.read_csv("data/feature_elimination1.csv")
```

[41]:
```
df3
```

[41]:

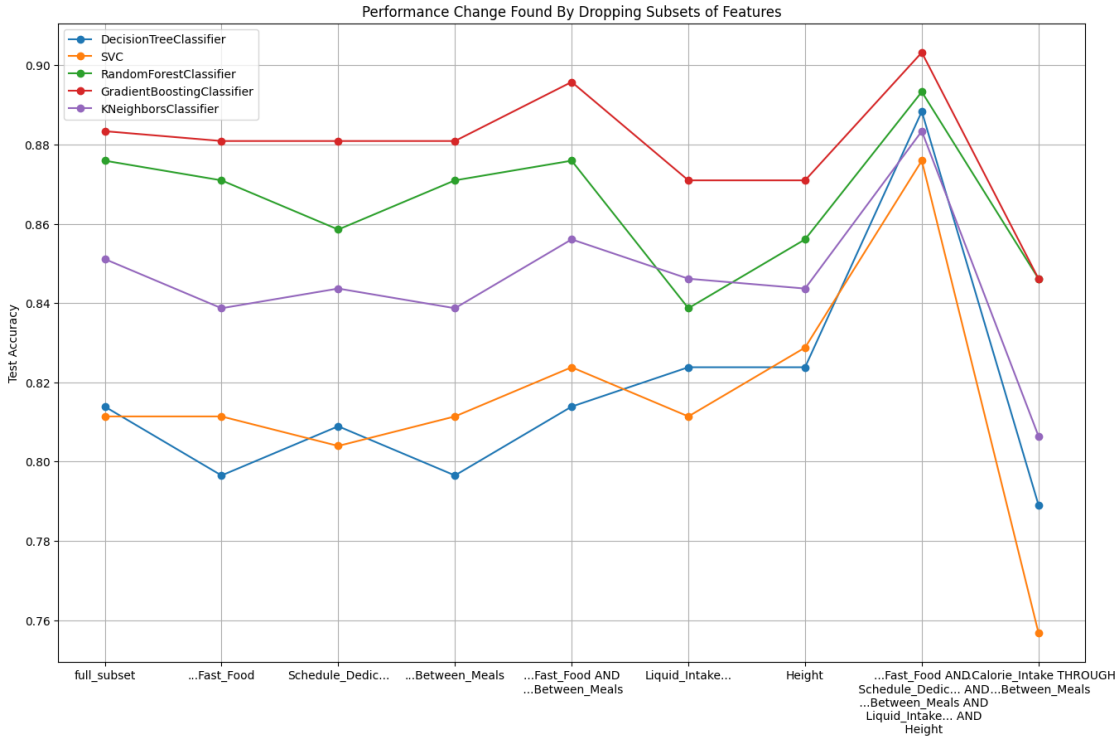
	classifier	full_mean_cv_score	full_train_score	full_test_score	ffbm_mean_cv_score	ffbm_train_score	ffbm_test_score
0	DecisionTreeClassifier	0.753952	0.961889	0.813896	0.761413	0.993372	0.813896
1	SVC	0.753930	0.995857	0.811414	0.756400	0.990058	0.823821
2	RandomForestClassifier	0.820207	0.997514	0.875931	0.827683	0.998343	0.875931
3	GradientBoostingClassifier	0.802815	1.000000	0.883375	0.814427	0.999171	0.895782
4	KNeighborsClassifier	0.778786	1.000000	0.851117	0.770503	0.999171	0.856079

```
   ks_mean_cv_score  ks_train_score  ks_test_score  …  h_test_score  \
0          0.729915        0.967688       0.789082  …      0.823821
1          0.697600        0.933720       0.756824  …      0.828784
2          0.792051        0.993372       0.846154  …      0.856079
3          0.763856        0.995029       0.846154  …      0.870968
4          0.719156        0.994200       0.806452  …      0.843672

   li_mean_cv_score  li_train_score  li_test_score  dt_mean_cv_score  \
0          0.748146        0.971831       0.823821          0.744824
1          0.752283        0.975973       0.811414          0.757258
2          0.818555        0.990886       0.838710          0.819371
3          0.814429        1.000000       0.870968          0.801990
4          0.768853        1.000000       0.846154          0.765544

   dt_train_score  dt_test_score  pba_mean_cv_score  pba_train_score  \
0        0.963546       0.808933           0.804484         0.962717
1        0.932063       0.803970           0.802810         0.968517
2        0.998343       0.858561           0.840936         0.974316
3        1.000000       0.880893           0.834291         0.974316
4        1.000000       0.843672           0.839264         0.974316

   pba_test_score
0        0.888337
1        0.875931
2        0.893300
3        0.903226
4        0.883375

[5 rows x 28 columns]
```

[42]:
```
cats = ['full_subset',
        '...Fast_Food',
        'Schedule_Dedic...',
        '...Between_Meals',
        '...Fast_Food AND\n ...Between_Meals',
        'Liquid_Intake...',
        'Height',
        '...Fast_Food AND\n Schedule_Dedic... AND\n ...Between_Meals AND\n␣
 ↪Liquid_Intake... AND\n Height',
        '...Calorie_Intake THROUGH\n ...Between_Meals']
vals = df3[['full_test_score',
           'ff_test_score',
           'dt_test_score',
           'bm_test_score',
           'ffbm_test_score',
           'li_test_score',
```

```
            'h_test_score',
            'pba_test_score',
            'ks_test_score']].T
plt.figure(figsize=(16, 10))
for i in range(len(classifier_names)):
    plt.plot(cats, vals[i], label=classifier_names[i], marker='o')
plt.grid(True)
plt.ylabel('Test Accuracy')
plt.legend()
plt.title('Performance Change Found By Dropping Subsets of Features')
```

[42]: Text(0.5, 1.0, 'Performance Change Found By Dropping Subsets of Features')



Interestingly, the individual elimination of Consumption_of_Fast_Food or Food_Intake_Between_Meals worsens outcomes after tuning for optimal hyperparameters, and yet eliminating both sees equivalent performance for DecisionTree and RandomForest while the rest improve somewhat. This can be improved further still by eliminating Schedule_Dedicated_to_Technology, Liquid_Intake_Daily, and Height as well. This maximizes performance for all the classifiers, save SVC (not quite at its best). The original hunch that the behavior seen in the out-of-the-box classifiers (earlier line chart) might lead to improved performance has seen some success.

0.1.9 Further Feature Elimination Analysis

Having used the methods for feature selection shown here followed by some model building, the quality of fitting for trained and parameter tuned models remained unsatisfactory. Because of this, further feature exploration work was performed under the feature_selection notebook included with the project. This work incorporated recursive feature elimination to examine accuracy changes for various feature inclusions, examining feature importance via RidgeCV coefficients, and exploring the loadings/coefficients of principal components derived from the data through PCA. Recursive feature elimination was helpful to reassure that the features included all showed improvement to mean test accuracy. Imporance via RidgeCV coefficents was generally helpful as a way of choosing which features to try excluding first, as it showed fairly different results than importance via MDI where RidgeCV considered different levels of regularization. PCA proved to be far more complicated. It serves as a different branch of exploratory analysis through which we can preprocess our original features in order to manage correlation and noise. It constructs a wholly new set of values for our existing observations to also be used with models, but can tell us about how datapoints regress within the feature space.

Considering these results in concert and checking resulting test performance for various additional eliminations proved unsuccessful. Testing accuracy declined for nearly all models with the additional exclusion of Smoking, Type_of_Transportation_Used, Overweight_Obese_Family, as well as combinations of all three and combining in pairs. The exception being that SVC apparently improves very slightly when also excluding Overweight_Obese_Family.

For the sake of the end performance comparison of the project these features will be excluded: Consumption_of_Fast_Food, Schedule_Dedicated_to_Technology, Liquid_Intake_Daily, Food_Intake_Between_Meals, and Height. However, were a significantly greater amount of training data to be collected, it might ultimatley prove beneficial to retain more features (or perhaps exclude others) depending on that influence. It is clear that for the dataset as it is, the information included by these features introduces greater noise to our models than they do predictive strength. This may have to do with a number of uncertain factors like the wording of those questions in the original survey or coincidence in behavior which obscures the relationship between these particular eating behaviors (or a person's height) and actual weight class.

[43]:
```
# reconfigure x_train, x_test to exclude Consumption_of_Fast_Food,␣
 ↪Schedule_Dedicated_to_Technology,
# Liquid_Intake_Daily, Food_Intake_Between_Meals, and Height

x_train = x_train.drop(['Consumption_of_Fast_Food',
                        'Schedule_Dedicated_to_Technology',
                        'Liquid_Intake_Daily',
                        'Food_Intake_Between_Meals',
                        'Height'], axis=1)

x_test = x_test.drop(['Consumption_of_Fast_Food',
                      'Schedule_Dedicated_to_Technology',
                      'Liquid_Intake_Daily',
                      'Food_Intake_Between_Meals',
                      'Height'], axis=1)
```

0.1.10 Still Room For Improvement!

Even with what success has been had, there is still significant potential for improvement with the dataset. This is illustrated very easily through evaluation metrics. As before, a DataFrame has been constructed to save time. The many evaluation metrics available through sklearn serve different purposes and better suit different data. It is also worthwhile to examine scoring results even when we only use default accuracy.

Where we use both accuracy and balanced accuracy with a dataset, we should expect to see very similar results in their scores when the classes being evaluated are correctly balanced. However, this is shown not to be the case when we examine the side-by-side bar charts below. Specifically, mean cross-validation score sits consistently below its counterpart with standard accuracy. This betrays the fact that classes are too imbalanced. Underweight respondents may not be underrepresented as a proportion to the population, but this would then mean that a metric which accomodates unbalanced multiple classes such as balanced accuracy ought to be used.

Another issue for the dataset illustrated here is how mean cross-validation score, training score, and testing score are so far apart from one another regardless of evaluation metric or classifier. This would seem to make for an easy clue about the overall quality of fit. It indicates that there is still overfitting being performed by high ranking parameters from GridSearchCV in order to fit the data to the classes. This score gap has improved noticeably from the original feature set and default parameters, but an uphill battle remains between classifiers and truly acceptable performance.

Using PCA makes for a new avenue of feature engineering, but for the sake of this early project I need to learn much more about it to introduce it effectively. Seeing what it could do with data was still helpful though. It lead me to the realization of how much potential further work could be done to produce a more effective solution even if we consider ourselves to have reached a hard limit with what existing features are able to tell us about the response we hope to predict. There is still plenty of room for improvement. The training process could definitely benefit from a dataset with more observations but also time spent further exploring feature engineering.

[44]:
```
df4 = pd.read_csv("data/evaluation_metrics1.csv")
df4
```

[44]:

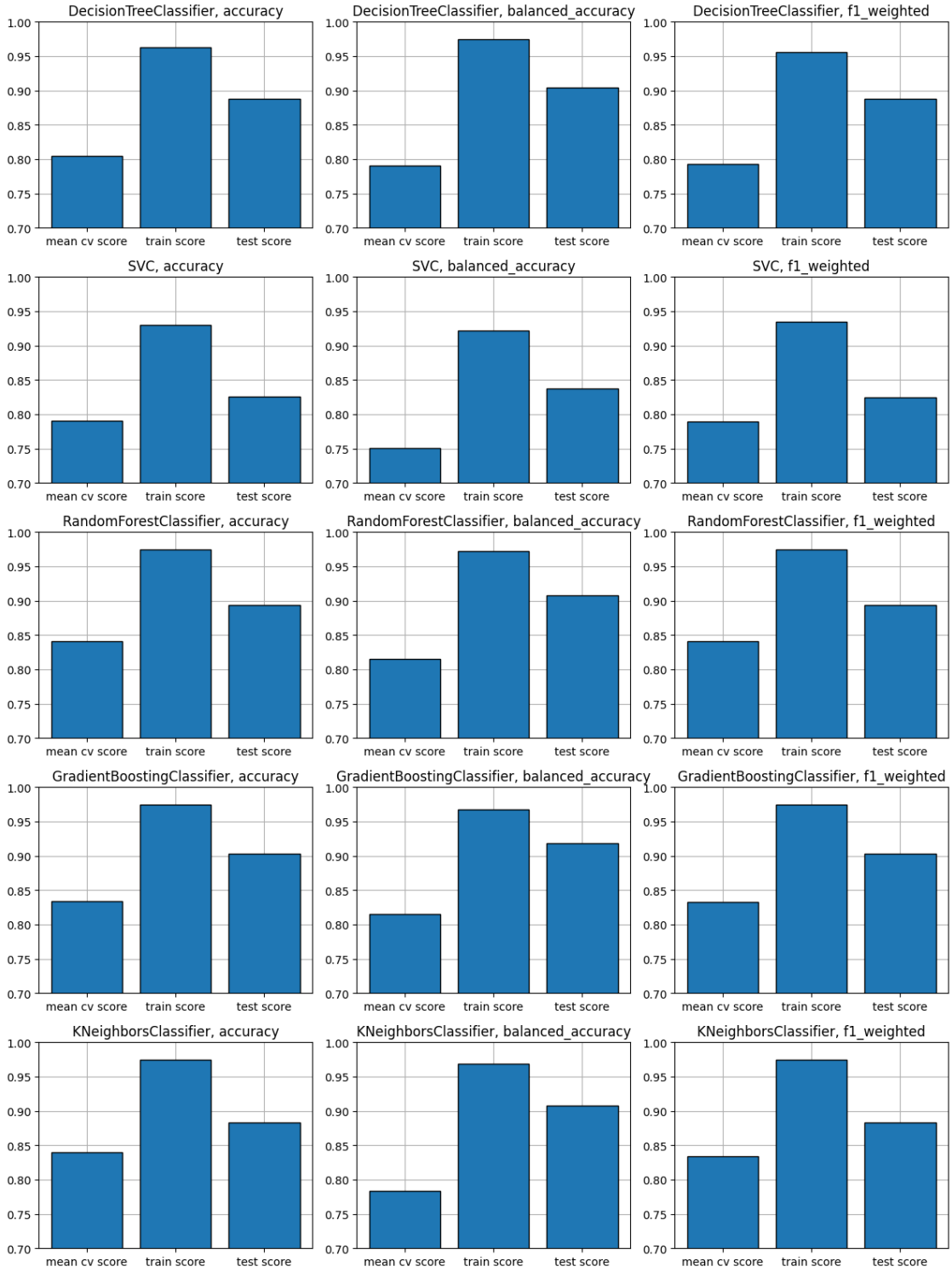
	DecisionTreeClassifier_acc	DecisionTreeClassifier_bal_acc	\
0	0.804484	0.791150	
1	0.962717	0.974458	
2	0.888337	0.904652	

	DecisionTreeClassifier_f1_w	SVC_acc	SVC_bal_acc	SVC_f1_w	\
0	0.793219	0.790398	0.750806	0.788961	
1	0.956173	0.930406	0.921736	0.935021	
2	0.888101	0.826303	0.836906	0.825001	

	RandomForestClassifier_acc	RandomForestClassifier_bal_acc	\
0	0.840936	0.815012	
1	0.974316	0.972259	
2	0.893300	0.908062	

```
   RandomForestClassifier_f1_w  GradientBoostingClassifier_acc  \
0                     0.840697                        0.834291
1                     0.974270                        0.974316
2                     0.893177                        0.903226

   GradientBoostingClassifier_bal_acc  GradientBoostingClassifier_f1_w  \
0                            0.815014                         0.832789
1                            0.967735                         0.974297
2                            0.918357                         0.903193

   KNeighborsClassifier_acc  KNeighborsClassifier_bal_acc  \
0                  0.839264                      0.783934
1                  0.974316                      0.968821
2                  0.883375                      0.907687

   KNeighborsClassifier_f1_w
0                   0.833713
1                   0.974247
2                   0.883123
```

[45]:
```
fig, ax = plt.subplots(5, 3, figsize=(12, 16))

x = ['mean cv score', 'train score', 'test score']
metrics = ['accuracy', 'balanced_accuracy', 'f1_weighted']
data = np.array(df4.T)

for i in range(5):
    for j in range(3):
        ax[i, j].grid(True)
        ax[i, j].bar(x, data[i*3+j], edgecolor='black', zorder=2)
        ax[i, j].set_title(f'{classifier_names[i]}, {metrics[j]}')
        ax[i, j].set_ylim(0.7, 1)


plt.tight_layout()
plt.show()
```


DecisionTreeClassifier, accuracy
DecisionTreeClassifier, balanced_accuracy
DecisionTreeClassifier, f1_weighted
SVC, accuracy
SVC, balanced_accuracy
SVC, f1_weighted
RandomForestClassifier, accuracy
RandomForestClassifier, balanced_accuracy
RandomForestClassifier, f1_weighted
GradientBoostingClassifier, accuracy
GradientBoostingClassifier, balanced_accuracy
GradientBoostingClassifier, f1_weighted
KNeighborsClassifier, accuracy
KNeighborsClassifier, balanced_accuracy
KNeighborsClassifier, f1_weighted
1.00
0.95
0.90
0.85
0.80
0.75
0.70
mean cv score
train score
test score

0.1.11 Model Choice Exploration

There is a wide variety of choices which apply to our classification problem. The model needs to support multiclass classification, provide a simple path to limit variance, and ideally works quickly. For the project, I have chosen five compatible models: decision trees, support vector machines, random forests, gradient boosted trees, and k-nearest neighbors. All of these are capable of performing fairly well for the data in the end, but all were subject to at least some hyperparameter values which demonstrate overfit.

0.1.12 Baseline Choice

I have chosen decision trees to serve as a baseline choice. It is simple and fast, making it a reasonable benchmark for the rest of the models.

1. Decision Trees

- **Type**: Nonparametric
- **Prediction**: Numeric (regression), Categorical (classification)
- **Hyperparameters**: Max depth, minimum samples per split/leaf, criterion (e.g., Gini, entropy)
- **Primal Function**: Tree-based decision rules (recursive partitioning)
- **Method**: Statistical (splits based on entropy, Gini index, etc.)
- **Variance Reduction**: Pruning the tree, limiting depth, or minimum samples per leaf

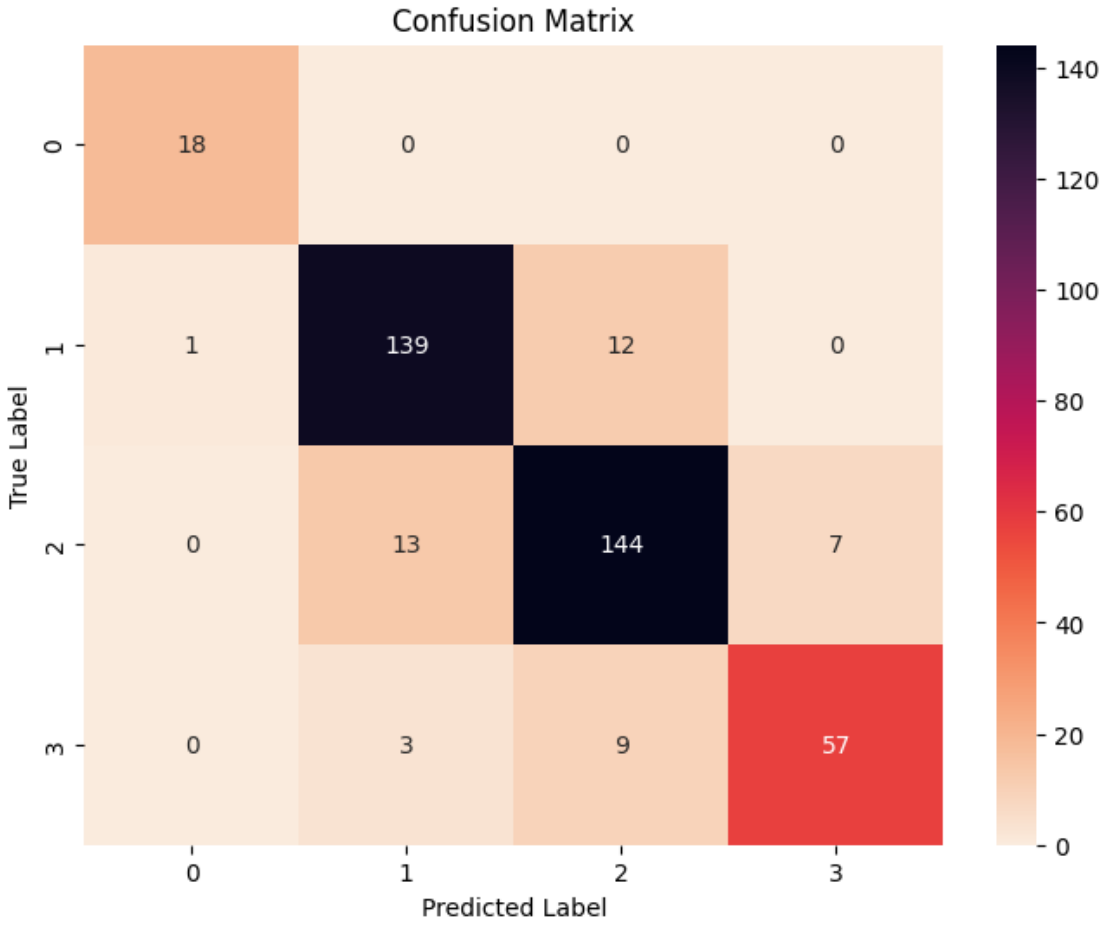
Decision trees use recursive binary splitting to grow a classification tree. Each node represents a decision rule and the leaves represent class outcomes. Decision trees are very simple or easy to understand, but they are subject to high variance and can often be inferior to more sophisticated methods.

However, a decision tree classifier will make a simple preliminary choice that will let us experiment quickly and easily, although it is generally less perfromant. The pipeline function defined above is used here and with the rest of the models in order to run through a grid of parameter options and have testing performance performed on the arrangements of parameters which have a higher cross-validation score.

[46]:
```
classifier = DecisionTreeClassifier


result_dict_decision_tree = pipeline(classifier,
                                     param_grids['DecisionTreeClassifier'],
                                     x_train,
                                     x_test,
                                     y_train,
                                     y_test,
                                     'balanced_accuracy')
```

```
{'ccp_alpha': 0, 'class_weight': 'balanced', 'criterion': 'gini', 'max_depth':
14, 'max_features': None, 'max_leaf_nodes': None, 'min_impurity_decrease': 0.0,
'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.0,
'random_state': 42, 'splitter': 'best'}
```

```
DecisionTreeClassifier Mean Cross-Validation score:  0.7911502905160459
DecisionTreeClassifier Training Score:  0.9744580298359157
DecisionTreeClassifier Testing Score:  0.9046523553050176
```



[47]:
```
pd.DataFrame(result_dict_decision_tree).sort_values('rank_test_score').head(3)
```

[47]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_ccp_alpha	param_class_weight	param_criterion	param_max_depth	param_random_state	params
27	0.003238	0.000060	0.001673	0.000016	0	None	entropy	14	42	{'ccp_alpha': 0, 'class_weight': None, 'criter…
41	0.003121	0.000052	0.001625	0.000035	0	None	log_loss	14		
13	0.002980	0.000073	0.001665	0.000037	0	None	gini	14		

```
41                42  {'ccp_alpha': 0, 'class_weight': None, 'criter…
13                42  {'ccp_alpha': 0, 'class_weight': None, 'criter…

    split0_test_score  split1_test_score  split2_test_score  \
27           0.801443           0.816032           0.781402
41           0.801443           0.816032           0.781402
13           0.799885           0.815179           0.789369

    split3_test_score  mean_test_score  std_test_score  rank_test_score
27           0.768557         0.791859        0.018224                1
41           0.768557         0.791859        0.018224                1
13           0.762320         0.791688        0.019280                3
```

We can see not only the parameter combination with best testing performance, but also how it scores and a representation of it's confusion matrix for the previously held out testing data. GridSearchCV returns a dictionary which we can use to examine what combinations of parameters performed well, and how similar thier cross-validated performance really was under 'mean_test_score'. This can inform us about what consistencies (or lack thereof) occur with parameter choices for the data and feature selection.

This decision tree behaves as would be expected. It is grown to be very tall, implying that it is overfitted to the training data. Typically we could aide this issue and improve the model's generalization by pruning the tree back by supplying a ccp_alpha parameter, dialing the max_depth back, or tuning min_samples_split to cut back on branch length. However, this final model is graded on test performance, meaning altering its parameters individually will be very likely to worsen misclassification for our given feature selection.

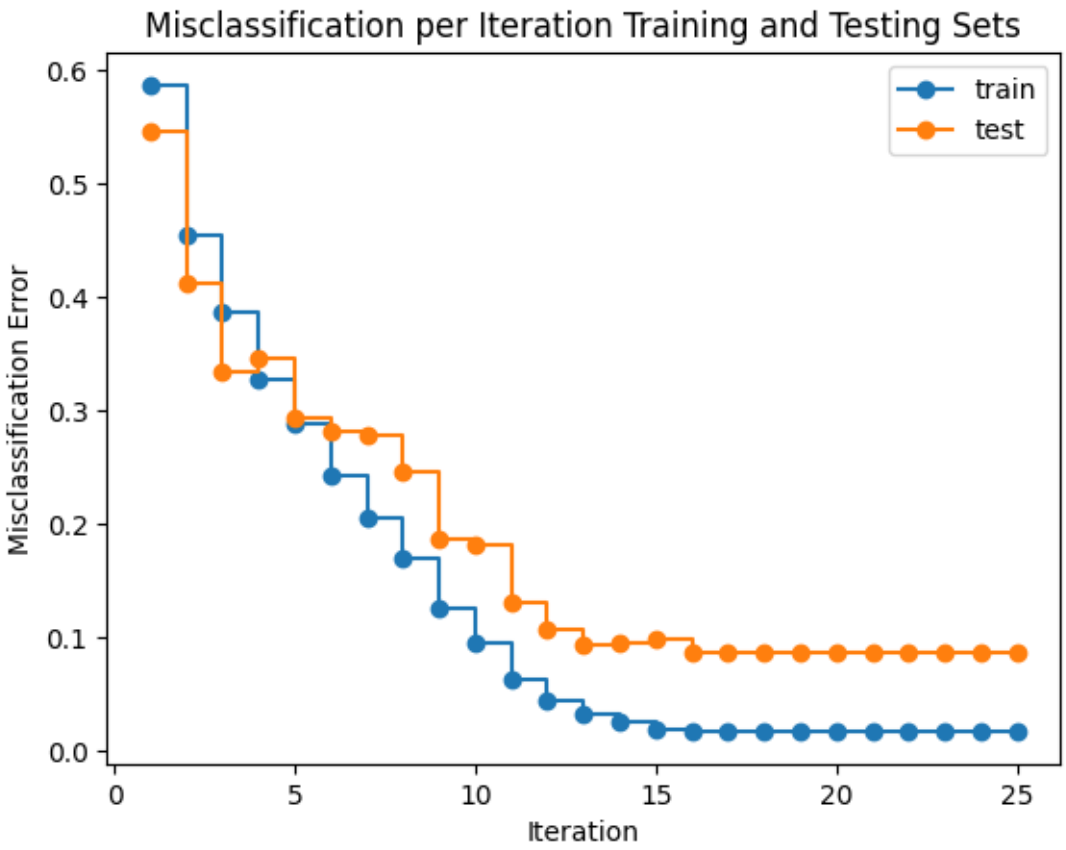
The misclass function is presented here, and is used to plot the training and testing misclassification error where we select a hyperparameter to tune, leaving other parameters in place. For our decision tree, error bottoms out around a depth of fourteen, and from here we find the model has about ten percent misclassification. Numerically this seems like okay performance, but inclines choosing the more complex random forests or gradient boosted trees because they have more robust parameter choices for controlling overfit.

This site was also very helpful to grow more comfortable with fit quality determinations with the kind of error plots we use.

[30]:
```
def misclass(x_tr, x_te, y_tr, y_te, p, params, clf=DecisionTreeClassifier,␣
 ↪n=1, N=5):
    misclass_train = []
    misclass_test = []

    for i in range(n,N+1):
        params[p] = i

        m = clf(**params).fit(x_tr, y_tr)

        y_hat_tr = m.predict(x_tr)
        y_hat_te = m.predict(x_te)
```

```
        train_score = balanced_accuracy_score(y_true=y_tr, y_pred=y_hat_tr)
        test_score = balanced_accuracy_score(y_true=y_te, y_pred=y_hat_te)

        misclass_train.append(1 - train_score)
        misclass_test.append(1 - test_score)

    x = np.array([range(n,N+1)])[0]

    fig, ax = plt.subplots()
    ax.set_xlabel("Iteration")
    ax.set_ylabel("Misclassification Error")
    ax.set_title("Misclassification per Iteration Training and Testing Sets")
    ax.plot(x, misclass_train, marker='o', label="train",
            drawstyle="steps-post")
    ax.plot(x, misclass_test, marker='o', label="test",
            drawstyle="steps-post")
    ax.legend()
    plt.show()
```

[31]:
```
param_DT = {'ccp_alpha': 0, 'class_weight': 'balanced', 'criterion': 'gini',␣
 ↪'max_depth': 14,
            'max_features': None, 'max_leaf_nodes': None,␣
 ↪'min_impurity_decrease': 0.0,
            'min_samples_leaf': 1, 'min_samples_split': 2,␣
 ↪'min_weight_fraction_leaf': 0.0,
            'random_state': 42, 'splitter': 'best'}

misclass(x_train,
         x_test,
         y_train,
         y_test,
         'max_depth',
         param_DT,
         DecisionTreeClassifier,
         n=1,
         N=25)
```



2. Support Vector Machines (SVM)

- **Type**: Parametric (non-linear SVM with kernel trick)
- **Prediction**: Categorical (classification), Numeric (regression - SVR)
- **Hyperparameters**: Regularization parameter `C`, kernel type (linear, polynomial, RBF), kernel coefficient `gamma`
- **Primal Function**: Maximizes the margin between classes (hinge loss)
- **Method**: Distance-based with kernel trick (if nonlinear)
- **Variance Reduction**: Regularization parameter `C` controls margin size (high `C` reduces bias, low `C` reduces variance)
- **Benefits from Regularization**: can benefit from standardiztion of feature values

Support vector machines are much more complicated but also have wider capabilities. It is a model that finds the optimal hyperplane (or decision boundary) that maximally separates classes by maximizing the margin between the closest data points (support vectors) of different classes. A datapoint's distance away from this boundary also helps quantify a "confidence" in it's classification.

Where decision trees can be predominately controlled by restrictions in depth or the metric used for splitting, support vector machines hyperparameters depend somewhat on 'kernels' to dictate what their underlying functions actually look like. All kernels available with SVC from sklearn feature a C hyperparameter, which works as a direct control for the amount of misclassification of

the model and resultingly how aggressively fit it becomes with training data. Nonlinear kernels also use a hyperparameter gamma. Gamma is a control to influence the extend to which more distant datapoints from the decision boundary are able to also influence it. Behavior in margins resulting from changes in gamma are subject to kernel choice.

Linear Kernel finds a straight hyperplane to separate classes in the original feature space.

Polynomial Kernel maps data to a higher-dimensional space using polynomial combinations of features, allowing non-linear decision boundaries.

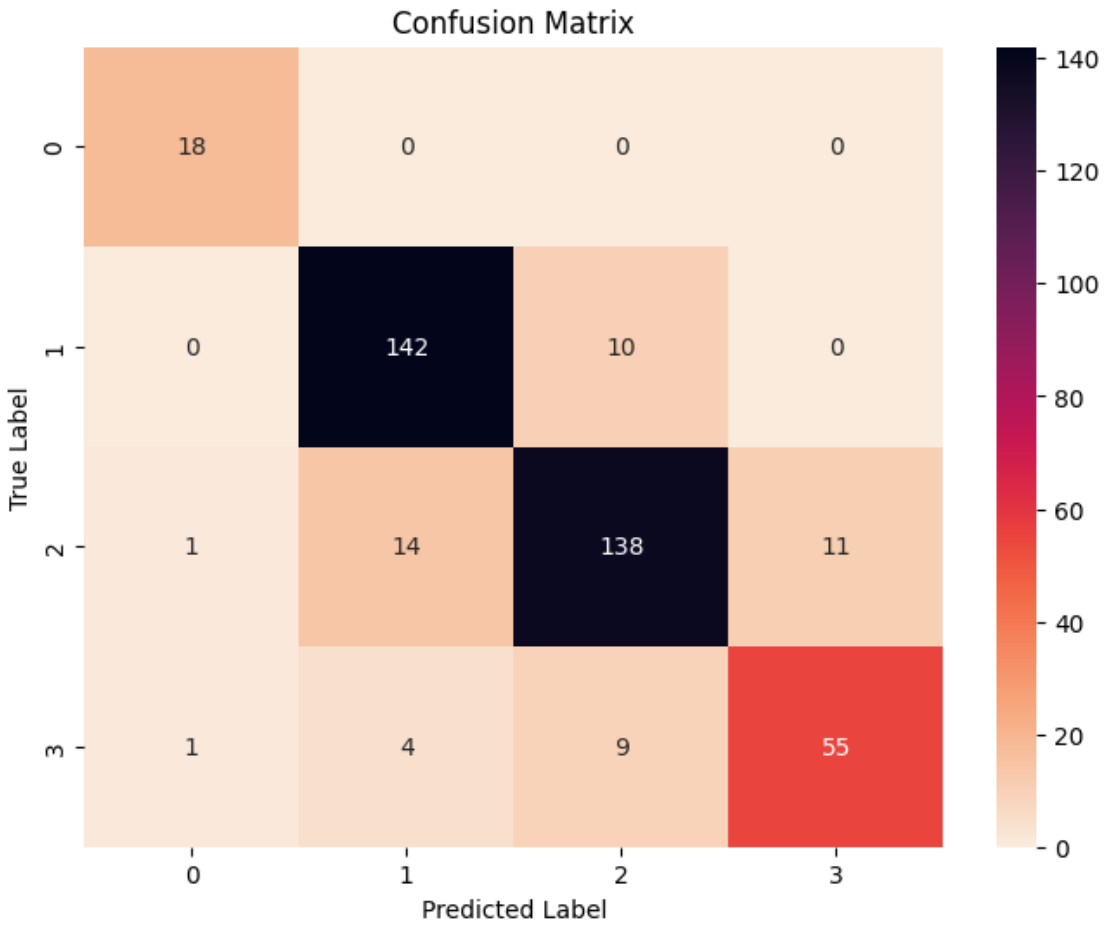
Radial Basis Function maps data points into an infinite-dimensional space based on their distance from each other, enabling highly flexible, non-linear decision boundaries.

Sigmoid Kernel maps data in a way that can model complex, non-linear relationships, based on the hyperbolic tangent function.

An online guid was helpful in reviewing how the SVC classifier works.

[23]:
```
classifier = SVC

x_tr_svc = x_train.copy()
x_te_svc = x_test.copy()

#x_tr_svc['Age'] = x_tr_svc['Age'] / np.linalg.norm(x_tr_svc['Age'], 1)
#x_te_svc['Age'] = x_te_svc['Age'] / np.linalg.norm(x_tr_svc['Age'], 1)

result_dict_svc = pipeline(classifier,
                           param_grids['SVC'],
                           x_tr_svc,
                           x_te_svc,
                           y_train,
                           y_test,
                           'balanced_accuracy')
```

```
{'C': 99, 'break_ties': False, 'cache_size': 200, 'class_weight': None, 'coef0':
0.0, 'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'auto', 'kernel':
'rbf', 'max_iter': -1, 'probability': False, 'random_state': 42, 'shrinking':
True, 'tol': 0.001, 'verbose': False}
SVC Mean Cross-Validation score:  0.7988379247720442
SVC Training Score:  0.9634664909366772
SVC Testing Score:  0.8931938475563245
```



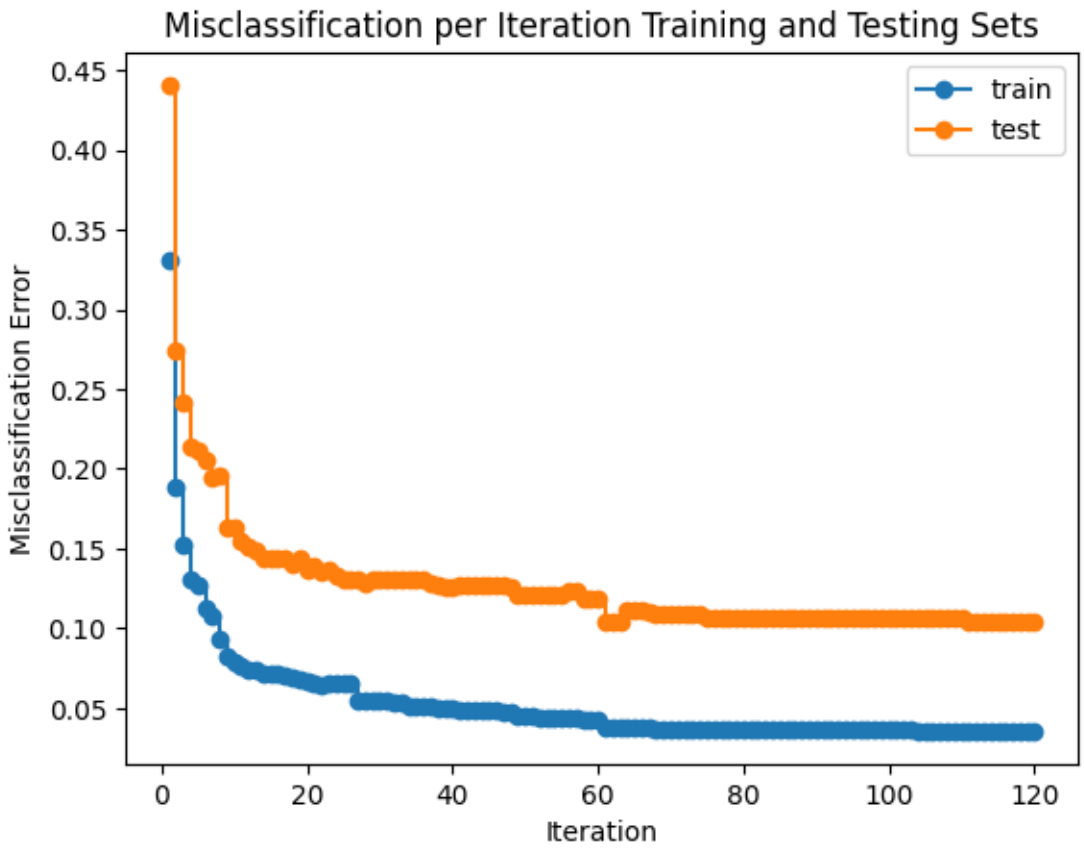
[8]:
```
pd.DataFrame(result_dict_svc).sort_values('rank_test_score').head(3)
```

[8]:
```
    mean_fit_time  std_fit_time  mean_score_time  std_score_time param_C  \
9        0.038649      0.002272         0.008261        0.000097      91
72       0.038815      0.001489         0.009158        0.001716      98
81       0.038517      0.000641         0.008194        0.000145      99

   param_gamma param_kernel param_random_state  \
9         auto          rbf                42
72        auto          rbf                42
81        auto          rbf                42

                                              params  split0_test_score  \
9   {'C': 91, 'gamma': 'auto', 'kernel': 'rbf', 'r…           0.800469
72  {'C': 98, 'gamma': 'auto', 'kernel': 'rbf', 'r…           0.795796
81  {'C': 99, 'gamma': 'auto', 'kernel': 'rbf', 'r…           0.795796

    split1_test_score  split2_test_score  split3_test_score  mean_test_score  \
```

```
9            0.734866           0.712561           0.69687          0.736191
72           0.734866           0.712561           0.69687          0.735023
81           0.734866           0.712561           0.69687          0.735023

    std_test_score  rank_test_score
9         0.039490                1
72        0.037595                2
81        0.037595                2
```

This svc model performs slightly worse than ten percent misclassification, so performance is only slightly worse than the decision tree. It's confusion matrix illustrates a recurring issue with the performance of all of our classifiers. Namely, precision and recall problems between the three larger weight classes. Still, performance has improved from out-of-the-box parameters. GridSearchCV consistently ranked the radial kernel highest, with a gamma value of 'auto' (equivalent to the inverse of the number of features available, in our case 1/9). Of the choices explored through GridSearchCV, the higher the C parameter was allowed to go the more it would, and tops out at ninety-nine. This seems extremely large. Although gamma influences aggressiveness of fit and is not large, because C specifically controlls for strength of fit this implies that the model will not generalize as well as it should.

[32]:
```
param_SVC = {'C': 99, 'break_ties': False, 'cache_size': 200, 'class_weight':␣
 ↪None, 'coef0': 0.0,
             'decision_function_shape': 'ovr', 'degree': 3, 'gamma': 'auto',␣
 ↪'kernel': 'rbf', 'max_iter': -1,
             'probability': False, 'random_state': 42, 'shrinking': True, 'tol':␣
 ↪0.001, 'verbose': False}

misclass(x_train,
         x_test,
         y_train,
         y_test,
         'C',
         param_SVC,
         SVC,
         n=1,
         N=120)
```



3. Random Forests

- **Type**: Nonparametric
- **Prediction**: Numeric (regression), Categorical (classification)
- **Hyperparameters**: Number of trees, max depth, minimum samples per leaf, number of features considered at each split
- **Primal Function**: Aggregates predictions from multiple decision trees (ensemble method)
- **Method**: Statistical (based on tree splits)
- **Variance Reduction**: reducing the subset of predictors used, increasing the number of trees

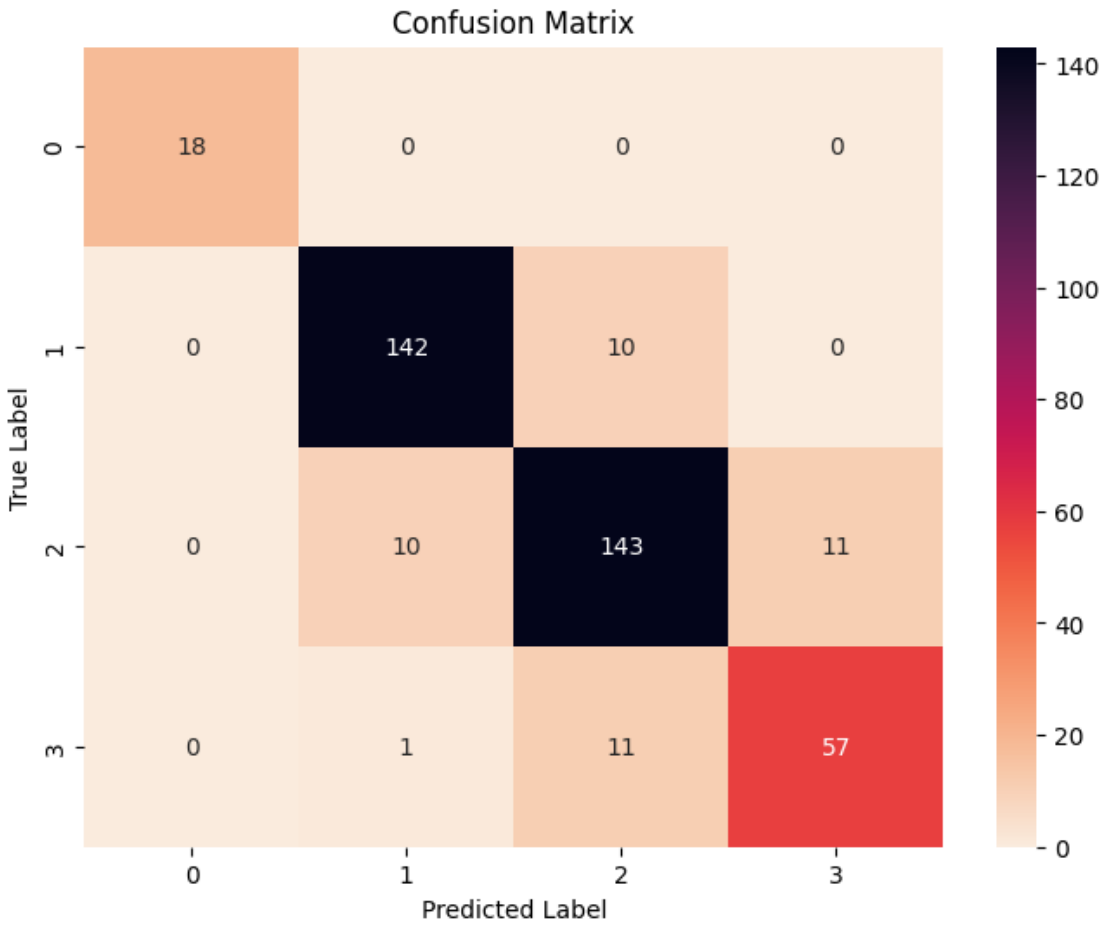
Random forests are the first of the ensemble methods. They are a way of averaging multiple decision trees, trained on different parts of the same training set, with the goal of reducing the variance. During training a series of trees are constructed using a bootstrap sample of the training observations and a randomized subset of the features of a predetermined size. Once a sufficient number of trees are constructed their individual predictions can be aggregated into a final result, using majority voting for classifications.

Random forests and ensembles in general are not as subject to variance as their individual tree counterparts. This is because the process of randomizing the features and observations used decorrelates their individual outcomes, and this has a collective effect of reducing the variance of their predictions. Random Forests still possess hyperparameters which can be used to control the charac-

teristics of their constituent trees, such as max_depth, but the approach generally used to minimize error for random forests takes advantage of the fact that increasing the total number of trees does not introduce further variance. The subset size of features in each tree is also used to improve variance, but can also increase bias.

[24]:
```
classifier = RandomForestClassifier

result_dict_random_forest = pipeline(classifier,
                                     param_grids['RandomForestClassifier'],
                                     x_train,
                                     x_test,
                                     y_train,
                                     y_test,
                                     'balanced_accuracy')
```

```
{'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None, 'criterion': 'gini',
'max_depth': 13, 'max_features': None, 'max_leaf_nodes': None, 'max_samples':
None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split':
2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 70, 'n_jobs': None,
'oob_score': False, 'random_state': 42, 'verbose': 0, 'warm_start': False}
RandomForestClassifier Mean Cross-Validation score:  0.8150115320979624
RandomForestClassifier Training Score:  0.9722593772271962
RandomForestClassifier Testing Score:  0.908062175587431
```



[132]:
```
pd.DataFrame(result_dict_random_forest).sort_values('rank_test_score').head(3)
```

[132]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	\
307	0.154703	0.002505	0.012416	0.000087	
331	0.155302	0.001624	0.012569	0.000226	
330	0.137327	0.003108	0.011474	0.000248	

	param_criterion	param_max_depth	param_max_features	param_n_estimators	\
307	gini	13	None	80	
331	gini	14	None	80	
330	gini	14	None	70	

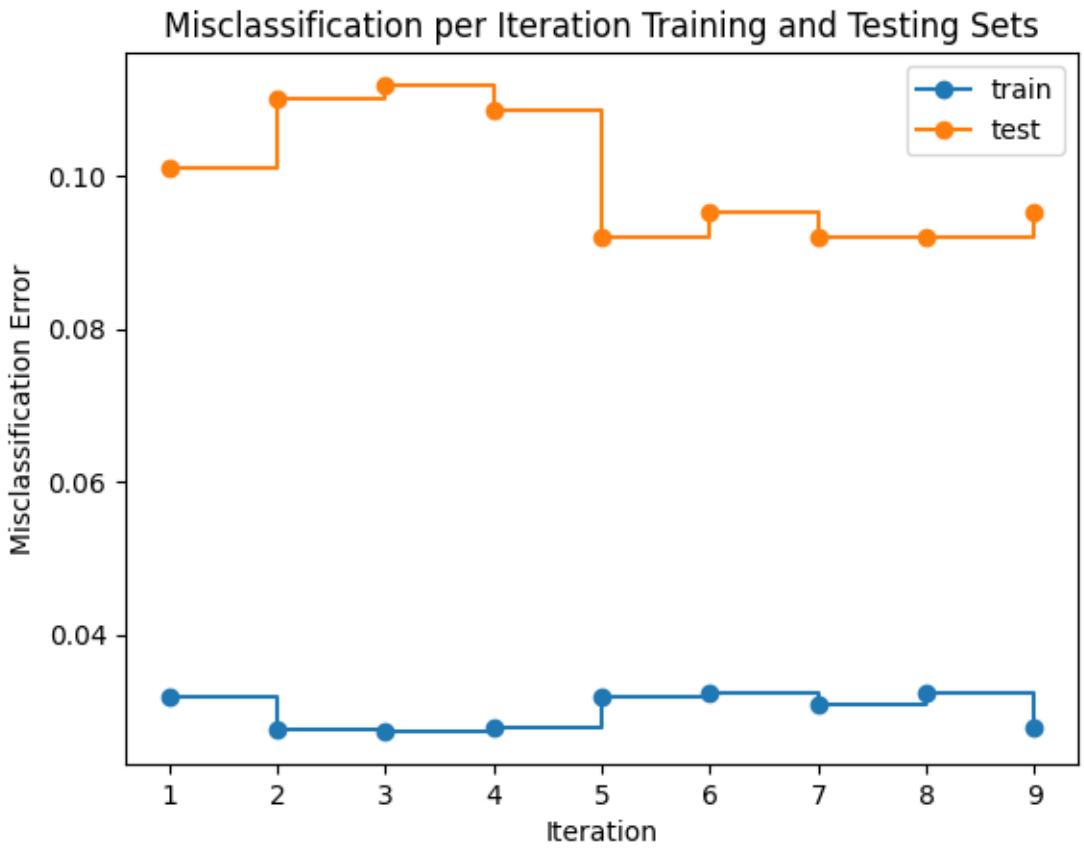
	param_random_state	params	\
307	42	{'criterion': 'gini', 'max_depth': 13, 'max_fe…	
331	42	{'criterion': 'gini', 'max_depth': 14, 'max_fe…	
330	42	{'criterion': 'gini', 'max_depth': 14, 'max_fe…	

	split0_test_score	split1_test_score	split2_test_score	\

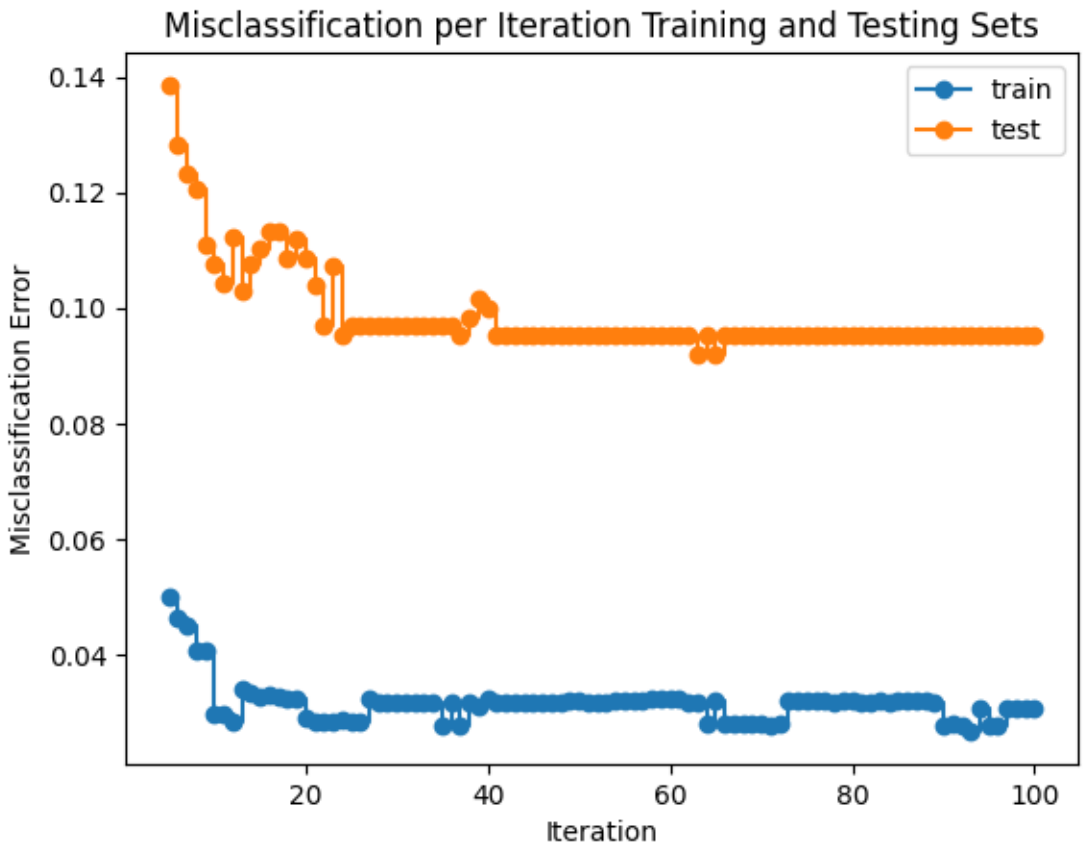
```
307           0.862913           0.817125           0.786907
331           0.862913           0.817818           0.780603
330           0.845055           0.824827           0.780603

     split3_test_score  mean_test_score  std_test_score  rank_test_score
307           0.814543         0.820372        0.027268                1
331           0.809998         0.817833        0.029494                2
330           0.809998         0.815121        0.023495                3
```

Unfortunately although random forests are not as subject to the effects of variance, this is not enough to significantly improve performance over individual trees for the dataset/feature selection. There is only slight improvement, despite the additional complexity and time required to construct the forest. Behavior shown by iterating through the possible choices of hyperparameters, namely the size of feature subsets used in tree construction, is fairly unexpected. Rather than performing better at a feature subset of around three (what would be the square-root of the total features) instead GridSearchCV ranks parameters higher when they use the full feature set, which would actually make these bagging ensembles rather than technically random forests. Also amiss is how misclassification error bottoms out very quickly. Rather than decreasing further, this training data inclines towards a model with much fewer overall trees and deeper trees than is typical for random forests.

[130]:
```
param_RF = {'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None,␣
 ↪'criterion': 'gini', 'max_depth': 14,
            'max_features': 9, 'max_leaf_nodes': None, 'max_samples': None,␣
 ↪'min_impurity_decrease': 0.0,
            'min_samples_leaf': 1, 'min_samples_split': 2,␣
 ↪'min_weight_fraction_leaf': 0.0, 'n_estimators': 70,
            'n_jobs': None, 'oob_score': False, 'random_state': 42, 'verbose':␣
 ↪0, 'warm_start': False}

misclass(x_train,
         x_test,
         y_train,
         y_test,
         'max_features',
         param_RF,
         RandomForestClassifier,
         n=1,
         N=9)
```



[129]:
```
param_RF = {'bootstrap': True, 'ccp_alpha': 0.0, 'class_weight': None,␣
 ↪'criterion': 'gini', 'max_depth': 14,
            'max_features': 9, 'max_leaf_nodes': None, 'max_samples': None,␣
 ↪'min_impurity_decrease': 0.0,
            'min_samples_leaf': 1, 'min_samples_split': 2,␣
 ↪'min_weight_fraction_leaf': 0.0, 'n_estimators': 50,
            'n_jobs': None, 'oob_score': False, 'random_state': 42, 'verbose':␣
 ↪0, 'warm_start': False}

misclass(x_train,
         x_test,
         y_train,
         y_test,
         'n_estimators',
         param_RF,
         RandomForestClassifier,
         n=5,
         N=100)
```



4. Gradient Boosting Machines (GBM)

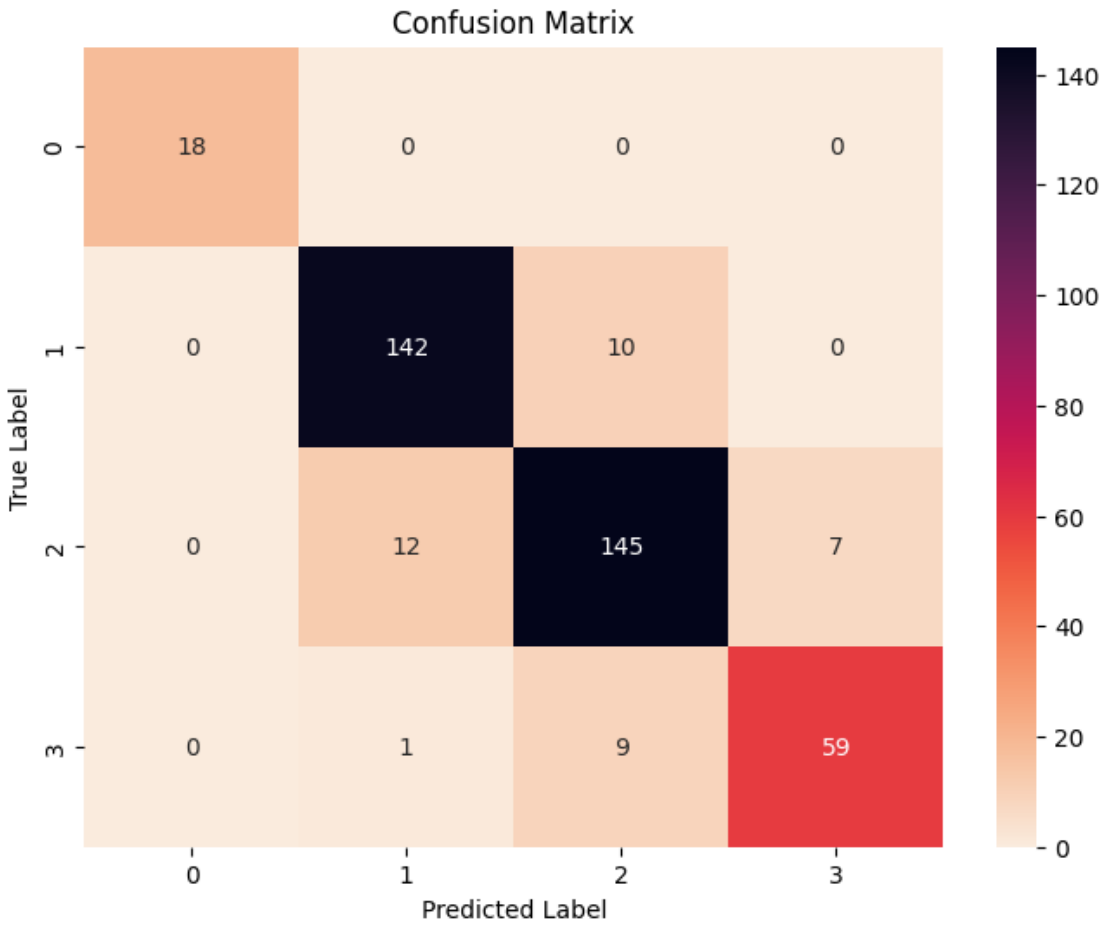
- **Type**: Nonparametric
- **Prediction**: Numeric (regression), Categorical (classification)
- **Hyperparameters**: Learning rate, number of trees, max depth, minimum samples per leaf, subsample ratio
- **Primal Function**: Sequentially builds trees, each correcting errors of the previous one (additive model)
- **Method**: Statistical (gradient descent on loss function)
- **Variance Reduction**: Reduced by limiting boosting iterations or tree depth
- **Benefits from Regularization**: Tuning the learning rate and subsampling rate can regularize the model

Another more complex esemble method is gradient boosting. It also works by combining the weaker learners of individual trees together to form a strong learner, only optimizing each subsequent learner in the series using the preexisting error of prior learners as an optimization problem through a procedure similar to gradient descent. Unlike random forests which constuct trees to be distinct and independent of one another, gradient boosted trees are built in stages where prior performance influences subsequent trees. Also unlike random forests, which use a number of trees deemed sufficiently large for error to reduce, gradient boosted trees can have their iterations (and subsequent number of trees) limited in order to positively influence variance.

gradient boosting has a regularization method known as shrinking, where the learning rate hyper-parameter is restricted to around 0.1 or less in order to help with generalization, although this results in a slower model. Learning rate can also be controlled for to help the model become more robust and avoid overfitting.

[25]:
```
classifier = GradientBoostingClassifier

result_dict_gradient_boost = pipeline(classifier,
                                      param_grids['GradientBoostingClassifier'],
                                      x_train,
                                      x_test,
                                      y_train,
                                      y_test,
                                      'balanced_accuracy')
```

```
{'ccp_alpha': 0.0, 'criterion': 'friedman_mse', 'init': None, 'learning_rate':
0.1, 'loss': 'log_loss', 'max_depth': 8, 'max_features': None, 'max_leaf_nodes':
None, 'min_impurity_decrease': 0.0, 'min_samples_leaf': 1, 'min_samples_split':
2, 'min_weight_fraction_leaf': 0.0, 'n_estimators': 140, 'n_iter_no_change':
None, 'random_state': 42, 'subsample': 1.0, 'tol': 0.0001,
'validation_fraction': 0.1, 'verbose': 0, 'warm_start': False}
GradientBoostingClassifier Mean Cross-Validation score:  0.8150138463700904
GradientBoostingClassifier Training Score:  0.9677353579899814
GradientBoostingClassifier Testing Score:  0.91835733288683
```



[135]:
```
# Previous result just in case
# {'ccp_alpha': 0.0, 'criterion': 'friedman_mse', 'init': None, 'learning_rate':
 ↪ 0.1, 'loss': 'log_loss',
# 'max_depth': 8, 'max_features': None, 'max_leaf_nodes': None,␣
 ↪'min_impurity_decrease': 0.0,
# 'min_samples_leaf': 1, 'min_samples_split': 2, 'min_weight_fraction_leaf': 0.
 ↪0, 'n_estimators': 140,
# 'n_iter_no_change': None, 'random_state': 42, 'subsample': 1.0, 'tol': 0.
 ↪0001, 'validation_fraction': 0.1,
# 'verbose': 0, 'warm_start': False}
# GradientBoostingClassifier Mean Cross-Validation score:  0.8150138463700904
# GradientBoostingClassifier Training Score:  0.9677353579899814
# GradientBoostingClassifier Testing Score:  0.91835733288683
#
pd.DataFrame(result_dict_gradient_boost).sort_values('rank_test_score').head(3)
```

[135]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	\
480	0.884558	0.019628	0.006505	0.000262	
342	0.954757	0.010990	0.006745	0.000122	
356	1.265937	0.010091	0.007778	0.000042	

	param_learning_rate	param_max_depth	param_n_estimators	param_random_state	\
480	0.4	8	50	42	
342	0.1	7	70	42	
356	0.1	8	70	42	

	params	split0_test_score	\
480	{'learning_rate': 0.4, 'max_depth': 8, 'n_esti…	0.882370	
342	{'learning_rate': 0.1, 'max_depth': 7, 'n_esti…	0.872571	
356	{'learning_rate': 0.1, 'max_depth': 8, 'n_esti…	0.849305	

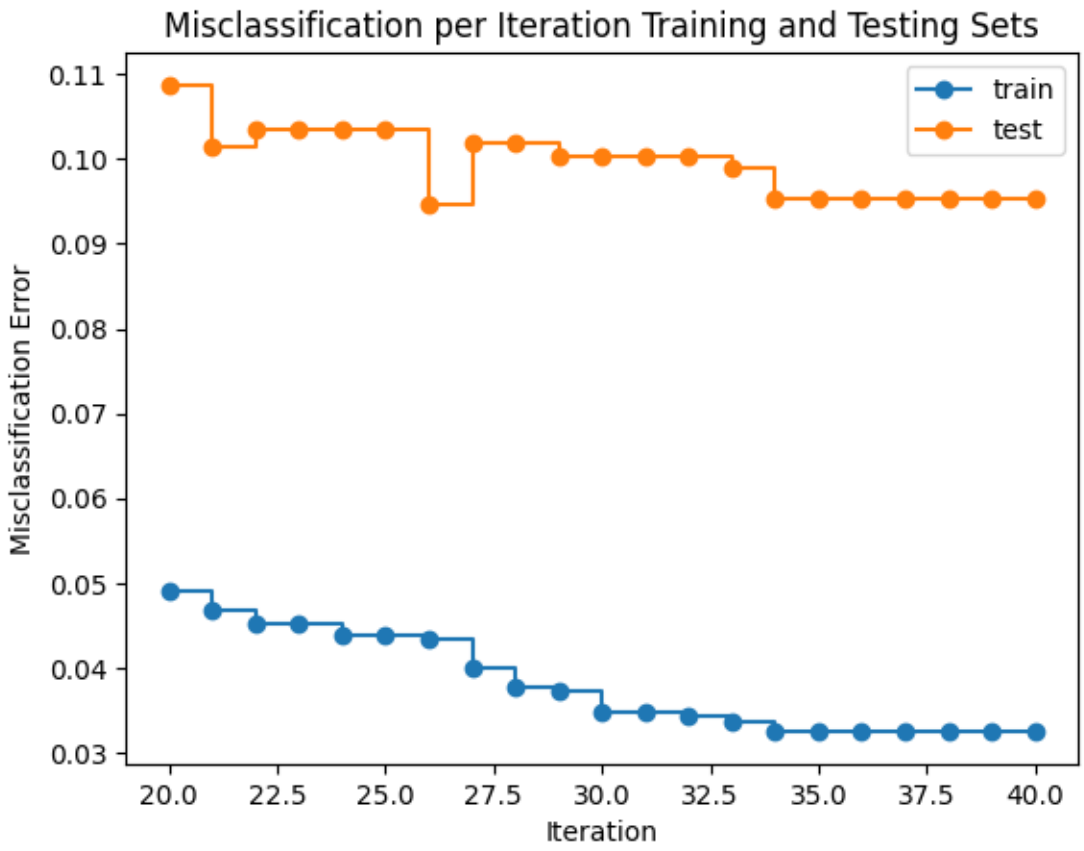
	split1_test_score	split2_test_score	split3_test_score	mean_test_score	\
480	0.816217	0.798914	0.775469	0.818243	
342	0.816131	0.816669	0.766378	0.817937	
356	0.823053	0.829856	0.766025	0.817060	

	std_test_score	rank_test_score
480	0.039748	1
342	0.037577	2
356	0.031000	3

Gradient boosting has shown to be the best performing model for the dataset, although again by a small margin compared to the previous model. We still retain similar tendency to miscalssify the three higher weight categories for one another, and although scoring values like mean cross-validation score have grown somewhat closer they remain distant from the training score. Ranking by GridSearchCV results in model parameters that do appear better than in prior models. For example, tree depth is now lower (around 8) and learning rate tends toward 0.1 which means that shrinking is helping to effect variance. However, number of estimators still appears high.

[33]:
```
param_GB = {'ccp_alpha': 0.0, 'criterion': 'friedman_mse', 'init': None,␣
 ↪'learning_rate': 0.1,
           'loss': 'log_loss', 'max_depth': 8, 'max_features': None,␣
 ↪'max_leaf_nodes': None,
           'min_impurity_decrease': 0.0, 'min_samples_leaf': 1,␣
 ↪'min_samples_split': 2,
           'min_weight_fraction_leaf': 0.0, 'n_estimators': 140,␣
 ↪'n_iter_no_change': None,
           'random_state': 42, 'subsample': 1.0, 'tol': 0.0001,␣
 ↪'validation_fraction': 0.1,
           'verbose': 0, 'warm_start': False}

misclass(x_train,
         x_test,
```

```
        y_train,
        y_test,
        'n_estimators',
        param_GB,
        GradientBoostingClassifier,
        n=20,
        N=40)
```



5. k-Nearest Neighbors (k-NN)

- **Type**: Nonparametric
- **Prediction**: Numeric (regression), Categorical (classification)
- **Hyperparameters**: Number of neighbors (`k`), distance metric (e.g., Euclidean, Manhattan)
- **Primal Function**: Distance-based (find the majority class or average value of nearest neighbors)
- **Method**: Distance-based (nonparametric)
- **Variance Reduction**: Increase the number of neighbors (`k`) reduces variance but may increase bias

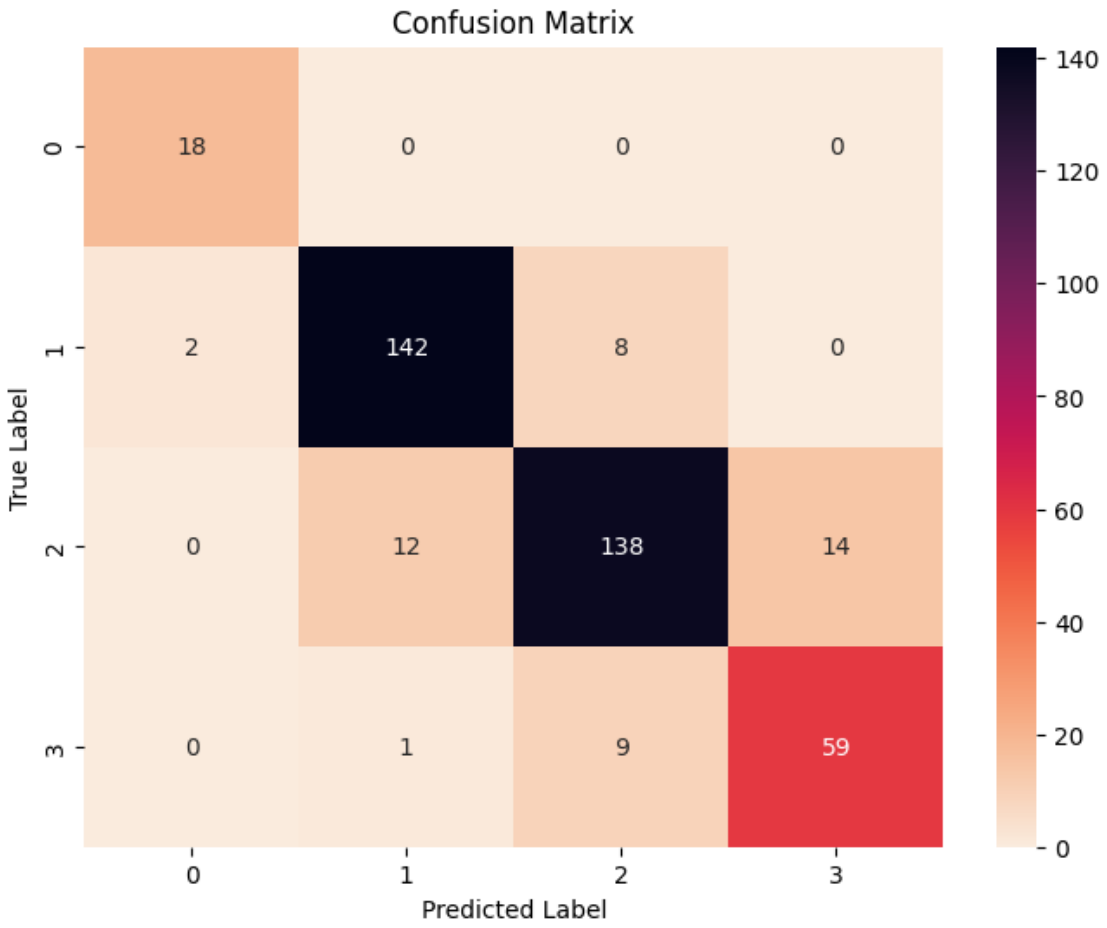
A conceptually simpler model choice is k-nearest neighbors. This method holds the training datapoints in memory, and performs a distance calculation to determine the subset which are calculated

closest to a testing datapoint before determining the prediction based on the majority vote of that subset as the prediction. The k hyperparameter controls the size of the 'neighborhood' used to determine a prediction, and also serves to control variance. It does this by consulting more datapoints (a larger neighborhood) in order to smooth out what the results will be, similar to how the gamma parameter allowed for less local datapoints to impose influence on prediction outcomes. With sklearn's implementation, KNeighborsClassifier, weights can also be introduced to influence classification by a factor of distance to aid less separable training data.

KNN is said to hold similarity to random forests. This similarity is in leveraging multiple decision boundaries through aggregation. KNN aggregates the class labels of nearby points (local neighbors) to make predictions, and random forests aggregate the predictions of many decision trees (each based on a random subset of data and features) to improve generalization.

[26]:
```
classifier = KNeighborsClassifier

result_dict_k_neighbors = pipeline(classifier,
                                   param_grids['KNeighborsClassifier'],
                                   x_train,
                                   x_test,
                                   y_train,
                                   y_test,
                                   'balanced_accuracy')
```

```
{'algorithm': 'brute', 'leaf_size': 30, 'metric': 'minkowski', 'metric_params':
None, 'n_jobs': None, 'n_neighbors': 1, 'p': 1, 'weights': 'distance'}
KNeighborsClassifier Mean Cross-Validation score:  0.7839341749370081
KNeighborsClassifier Training Score:  0.9688213189983933
KNeighborsClassifier Testing Score:  0.9076866011795128
```



[118]:
```
pd.DataFrame(result_dict_k_neighbors).sort_values('rank_test_score').head(3)
```

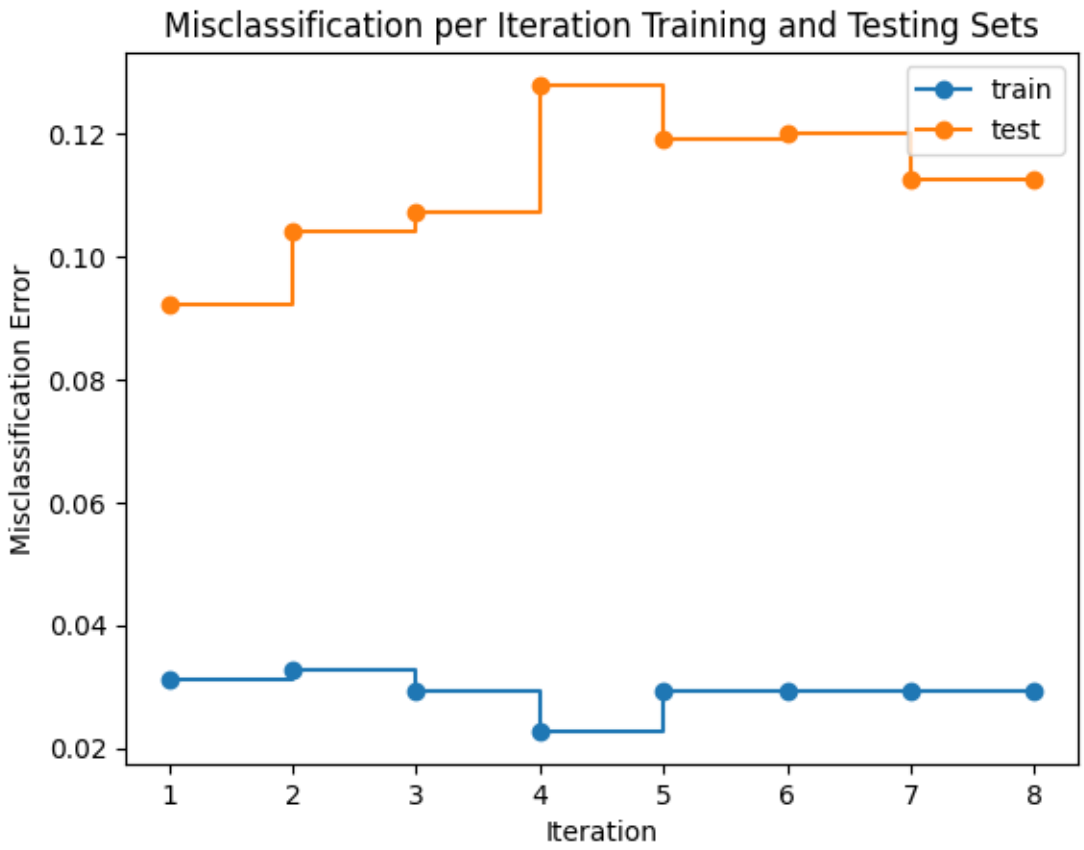
[118]:

```
     mean_fit_time  std_fit_time  mean_score_time  std_score_time  \
156       0.001726      0.000200         0.007633        0.000188
4         0.002211      0.000155         0.003884        0.000082
80        0.002207      0.000047         0.003639        0.000047

    param_algorithm param_n_neighbors param_p param_weights  \
156           brute                 2       1      distance
4         ball_tree                 2       1      distance
80          kd_tree                 2       1      distance

                                                params  split0_test_score  \
156  {'algorithm': 'brute', 'n_neighbors': 2, 'p': …           0.844949
4    {'algorithm': 'ball_tree', 'n_neighbors': 2, '…           0.824822
80   {'algorithm': 'kd_tree', 'n_neighbors': 2, 'p'…           0.829132

     split1_test_score  split2_test_score  split3_test_score  mean_test_score  \
```

```
156           0.786528           0.781511           0.783880           0.799217
4             0.790581           0.775283           0.799659           0.797586
80            0.790581           0.770935           0.795928           0.796644

     std_test_score  rank_test_score
156        0.026463                1
4          0.017976                2
80         0.020938                3
```

The main issue with our ranking outcome for k-nearest is the low number of neighbors. Where we see best testing performance with only one neighbor used, it seems assured that our feature set needs to be changed, as we are experiencing overfitting. It may be then that our model is only able to fit to this extent because its distance calculations result in a neighborhood which muddies results due to how datapoints of a given class are not producing a consistent majority for their positions within the feature space. We can see from the train and test accuracy plot below how performance changes with choice of k.

[138]:
```
param_KN = {'algorithm': 'brute', 'leaf_size': 30, 'metric': 'minkowski',␣
 ↪'metric_params': None,
            'n_jobs': -1, 'n_neighbors': 1, 'p': 1, 'weights': 'distance'}

misclass(x_train,
         x_test,
         y_train,
         y_test,
         'n_neighbors',
         param_KN,
         KNeighborsClassifier,
         n=1,
         N=8)
```



0.1.13 Model Results

code to create multiclass ROC curve plots for each final model

In order to make a comparison with the various models we needed a way to consider more than individual scores. The plot of the ROC curve makes for a great choice, because it summarizes two error rates (False Positive rate and True Positive Rate) across the possible thresholds. Traditionally, they are used for binary classification and we would use ROC AUC in order to plot multiple models under the same plot, but because our problem is a multiclass one we can instead plot the classifiers side by side. Be sure to double click the plots to see a more zoomed in view for close comparison.

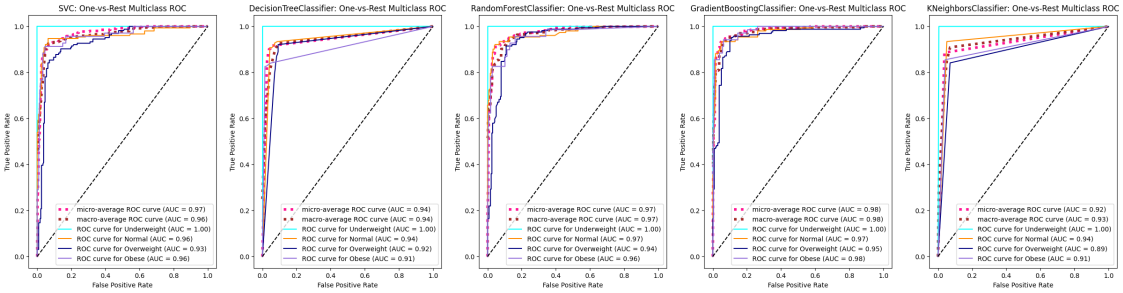
The method used to make ROC AUC work for multiclass models was found through sklearn. It uses a "One-vs-Rest" approach to determine the errors of each class for a given theshold. We also include micro and macro averaging for the classes, where macro averaging is an unweighted average across all the classes and micro averaging aggregates the individual error contributions from all the classes. We see from sklearn that a multi-class classification setup with highly imbalanced classes, micro-averaging is preferable over macro-averaging. Meaning the curve produced by the micro average may be somewhat superior in our case. In this way, we have the ability to interpret behavior of our model's between classes as well as collectively.

[27]:
```
models_list = list(models_dict.keys())
names = ['Underweight','Normal','Overweight','Obese']
K = len(models_list)
n_classes = 4

if 'SVC' in models_list:
    models_dict['SVC']['params']['probability'] = True

fig, ax = plt.subplots(1, 5, figsize=(30, 7))

for k in range(K):

    classifier = models_dict[models_list[k]]['classifier']
    params = models_dict[models_list[k]]['params']

    y_score = classifier(**params).fit(x_train, y_train).predict_proba(x_test)

    label_binarizer = LabelBinarizer().fit(y_train)
    y_onehot_test = label_binarizer.transform(y_test)
    y_onehot_test.shape  # (n_samples, n_classes)

    # store the fpr, tpr, and roc_auc for all averaging strategies
    fpr, tpr, roc_auc = dict(), dict(), dict()
    # Compute micro-average ROC curve and ROC area
    fpr["micro"], tpr["micro"], _ = roc_curve(y_onehot_test.ravel(), y_score.
 ↪ravel())
    roc_auc["micro"] = auc(fpr["micro"], tpr["micro"])


    for i in range(n_classes):
        fpr[i], tpr[i], _ = roc_curve(y_onehot_test[:, i], y_score[:, i])
        roc_auc[i] = auc(fpr[i], tpr[i])

    fpr_grid = np.linspace(0.0, 1.0, 1000)

    # Interpolate all ROC curves at these points
    mean_tpr = np.zeros_like(fpr_grid)

    for i in range(n_classes):
        mean_tpr += np.interp(fpr_grid, fpr[i], tpr[i])  # linear interpolation

    # Average it and compute AUC
    mean_tpr /= n_classes

    fpr["macro"] = fpr_grid
    tpr["macro"] = mean_tpr
    roc_auc["macro"] = auc(fpr["macro"], tpr["macro"])
```

```
    macro_roc_auc_ovr =␣
 ↪roc_auc_score(y_test,y_score,multi_class="ovr",average="macro",)

    print(f"Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for␣
 ↪{models_list[k]} \
classifier: \n{roc_auc['micro']:.2f}, {macro_roc_auc_ovr:.2f}")

    ax[k].plot(fpr["micro"],tpr["micro"],label=f"micro-average ROC curve (AUC =␣
 ↪{roc_auc['micro']:.2f})",
        color="deeppink",linestyle=":",linewidth=4,
    )

    ax[k].plot(fpr["macro"],tpr["macro"],label=f"macro-average ROC curve (AUC =␣
 ↪{roc_auc['macro']:.2f})",
        color="brown",linestyle=":",linewidth=4,
    )

    colors = ["aqua", "darkorange", "navy", "mediumpurple"]
    for class_id, color in zip(range(n_classes), colors):
        RocCurveDisplay.from_predictions(
            y_onehot_test[:, class_id],
            y_score[:, class_id],
            name=f"ROC curve for {names[class_id]}",
            color=color,
            ax=ax[k],
        )

    _ = ax[k].set(xlabel="False Positive Rate",ylabel="True Positive Rate",
        title=f"{models_list[k]}: One-vs-Rest Multiclass ROC",
    )

    ax[k].plot([0, 1], [0, 1], 'k--', label='Chance level')
```

```
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for SVC classifier:
0.97, 0.96
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for
DecisionTreeClassifier classifier:
0.94, 0.94
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for
RandomForestClassifier classifier:
0.97, 0.97
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for
GradientBoostingClassifier classifier:
0.98, 0.98
Micro-averaged and Macro-Averaged One-vs-Rest ROC AUC score for
KNeighborsClassifier classifier:
```

```
0.92, 0.93
```



0.1.14 Good Area? Good Steepness?

The decision tree classifier eventually developed fairly competitive accuracy for test data, but we can see things compare rather differently with the ROC plot. Its simplicity may be at hand for why false positives are noticeably worse than more complex methods. For earlier threshold values the decision tree shows steep improvement, but this suddenly gives way to more false positives. Consequently the plot very triangular.

We see that the svm, which uses the radial kernel, fairs better than decision tree. It does a better job of making gains in true positive rate for increased thresholds. However, like we see with the ensemble methods its overweight curve is less consistent. This corresponds with results from the confusion matrices of the models, which show that class to be regularly confused with others. The micro and macro averages of the svm are greater than the decision tree, helping let us know that the area is greater and thereby performance is higher.

For the two ensemble methods of random forest and gradient boosted trees ROC is extremely similar and their performances are the best overall. They have the greatest area and round off at a fairly high True Positive value. Gradient boosted trees win out slightly due to what appears to be better performance with the overweight category curve; bringing it further in line with the other classes. This slight edge comes at a cost though, as it was easily the slowest to train and predict with.

Finally, the k-neighbors classifier comes in last by a distinct margin with ROC. Similar to decision trees the k-neighbors plot has generally worse steepness and area, creating a clear triangle shape - only with even less consistency to the curves of its different classes.

0.1.15 Wrapping Up

We saw in the ROC plots that the overweight class in particular performed the worst for most of the models. It is also surprising how all the models manage to always learn to correctly identify all true cases of the underweight class. Being able to see results in this way, which allow focus on individual classes helps clarify how certain models are outperforming others. The decision tree model is growing extremely tall to involves enough splitting to compensate for overlaping qualities of the underlying datapoints. This brute forces enough sufficiently pure leaves to get somewhere.The worst performer, the k-neighbors model, is built on distance. Where distancing between datapoints of differing classes becomes too heavily mixed, it would make sense that performance by class would

be inconsistent. This would also lead an optimization search in the direction of fewer neighbors for classification, because very near neighbors belong to another class which muddies results. The radial kernel for svm is also influenced by intermixed datapoints, but its non-linear nature clearly lends enough flexibility to climb its way out of the hole of which k-neighbors cannot. Although this was only possible with a very heavy C value.

Likewise ensemble methods use greater complexity specifically because it can aid in stratifying heavily mixed spaces of datapoints. Random forests use of bagging, meaning that only a randomized subset of datapoints are considered for any individual tree, would naturally avoid some of the intermixing by randomly cutting some of it out - though this effect is slim. This also helps explain why the parameter ranking process of GridSearchCV prioritized as many features as possible for random forests in our case, because adding dimensionality was helping to diversify the interpretability of the datapoints in a way that the ensemble could not get through its typical behavior of having very many trees. The boosted decision trees model was likely then winning out by not needing to use trees of such a significant depth (not needing to split datapoints as aggressively) but having a very large number of overall trees/iterations and taking a slower learning rate. This gave the boosted ensemble very many iterations to weed out its mistakes until inevitably correcting for as much error as it could.

For now a choice of which makes for the best model is both clear and not. Based on the balanced accuracy and ROC curve area we found, for now we should probably just pick gradient boosting for the best model. Were the overfitting not the case, we would expect sginificantly better performance in comparison to a "weak" learner like the decision tree, which still manages disatisfyingly similar performance. However, the necessary conclusion is far more likely that there is still much work to be done in order to produce a more predictive feature set. Importantly, I believe that where it were not necessary for these models to have to overfit in each case, we would be able to see more diversification in their individual performance. Which would make selection of best model more obvious.

Although we can say that the features we have selected for the data up to this point provide some predictive ability for our classifier models, the nature of the datapoints is that they are still sufficiently indiscernable to impose hard limits of how effectively models can learn them. What predictive power the models can achieve, which is at least okay, universally requires that they overfit to perform their best. Where datapoints were sufficiently separable, then models would be able to learn the data faster and testing and cross-validation performance would be able to more closely resemble the training performance achieved. This might be helped by features which involve greater cardinality, to diversify their distributions, or otherwise including more predictive numeric attributes.

The project had some successes. My hunch about trendline behavior for selecting feature eliminations to improve the results of parameter tuning and testing performance really did work. Although learning about additional techniques like recursive feature elimination and PCA made for valuable experience to further explore machine learning, though I did not think that my experience or remaining time was sufficient to explore and include it properly here. Under that limitation the stubborn nature of fitting models made for an important lesson about overfitting, and helped me to better understand the extent to which models may exploit their flexibility to still achieve some prediction. Moving forward though, an obvious direction to explore would be learning more about feature engineering. It would help situations where we believe features can offer better predictive performance, but may need to be altered or combined differently to allow for the models to learn more easily and prevent variance.

0.1.16 References

1. Kaggle Dataset: Sulak, S. (n.d.). *Obesity Dataset*. Kaggle. Retrieved from https://www.kaggle.com/datasets/suleymansulak/obesity-dataset?resource=download
2. Forest Importances Example (Scikit-Learn): Scikit-learn. (n.d.). *Plot Feature Importance Using a Random Forest*. Retrieved from https://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_importances.html
3. Grid Search CV Documentation (Scikit-Learn): Scikit-learn. (n.d.). *sklearn.model_selection.GridSearchCV*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html
4. Learning Curves (Machine Learning Mastery): Brownlee, J. (2019, August 7). *Learning Curves for Diagnosing Machine Learning Model Performance*. Machine Learning Mastery. Retrieved from https://machinelearningmastery.com/learning-curves-for-diagnosing-machine-learning-model-performance/
5. Guide to SVM (NTU): Lin, C.-J. (n.d.). *A Practical Guide to Support Vector Classification*. National Taiwan University. Retrieved from https://www.csie.ntu.edu.tw/%7Ecjlin/papers/guide/guide.pdf
6. ROC Example (Scikit-Learn): Scikit-learn. (n.d.). *Receiver Operating Characteristic (ROC) with Cross Validation*. Retrieved from https://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html
7. Decision Tree Classifier Documentation (Scikit-Learn): Scikit-learn. (n.d.). *sklearn.tree.DecisionTreeClassifier*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html
8. SVC Documentation (Scikit-Learn): Scikit-learn. (n.d.). *sklearn.svm.SVC*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
9. Random Forest Classifier Documentation (Scikit-Learn): Scikit-learn. (n.d.). *sklearn.ensemble.RandomForestClassifier*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
10. Gradient Boosting Classifier Documentation (Scikit-Learn): Scikit-learn. (n.d.). *sklearn.ensemble.GradientBoostingClassifier*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html
11. K-Neighbors Classifier Documentation (Scikit-Learn): Scikit-learn. (n.d.). *sklearn.neighbors.KNeighborsClassifier*. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html
12. RFE with Cross-Validation Example (Scikit-Learn): Scikit-learn. (n.d.). *Recursive Feature Elimination with Cross-Validation*. Retrieved from https://scikit-learn.org/stable/auto_examples/feature_selection/plot_rfe_with_cross_validation.html
13. SelectFromModel Example (Scikit-Learn): Scikit-learn. (n.d.). *Feature Selection using SelectFromModel and LassoCV*. Retrieved from https://scikit-learn.org/stable/auto_examples/feature_selection/plot_select_from_model_diabetes.html
14. PCA Documentation (Scikit-Learn): Scikit-learn. (n.d.).

sklearn.decomposition.PCA. Retrieved from https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[ ]: