Final Project

March 16, 2024

[ ]:
```
# Training with M1 GPU
import tensorflow as tf
import os
print(tf.config.list_physical_devices('GPU'))
os.environ['CUDA_VISIBLE_DEVICES'] = '1'
```

```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

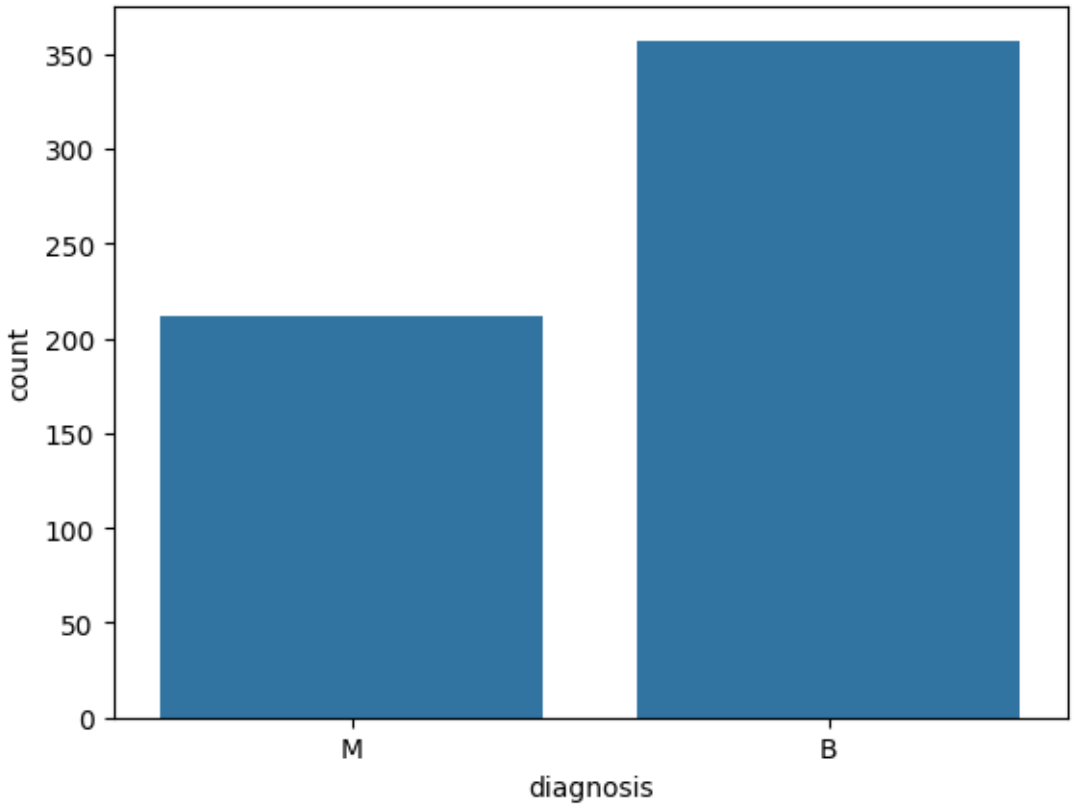
[ ]:
```
# IMPORTING LIBRARIES

import matplotlib.pyplot as plt # Plotting
from mpl_toolkits.mplot3d import Axes3D # 3D plotting
import tensorflow as tf # Tensors and AI utilities
import seaborn as sns # Plotting
import pandas as pd # Data manipulation
import numpy as np # Linear algebra

from sklearn.model_selection import train_test_split, RandomizedSearchCV #␣
 ↪Cross validation tools
from sklearn.tree import plot_tree # Decision tree visualization
from sklearn.metrics import confusion_matrix, f1_score, accuracy_score,␣
 ↪log_loss # Results metrics and visualization
from sklearn.utils import resample # Data balancing

from sklearn.ensemble import VotingClassifier as VC # Voting Classifier
from sklearn.neighbors import KNeighborsClassifier as KNC # Clustering
from sklearn.linear_model import LogisticRegression as LR # Logistic regression
from sklearn.tree import DecisionTreeClassifier as DTC # Decision Tree
from sklearn.ensemble import RandomForestClassifier as RFC # Random forest
from sklearn.ensemble import AdaBoostClassifier as ABC # Adaptive boosting
from sklearn.ensemble import GradientBoostingClassifier as GBC # Gradient␣
 ↪boosting
from xgboost import XGBClassifier as XBC # Extreme gradient boosting

from sklearn.base import BaseEstimator, ClassifierMixin

from keras.layers import (
```

```
    Input, # Input layer
    Flatten, # Dimension flattening
    Dense, # Densely connected layer
    LeakyReLU, # Activation
    Softmax, # Probability activation (final)
    Conv2D, # Convolutional filters
    BatchNormalization, # Normalization
    Dropout, # Regularization
    MaxPool2D # Maximum pooling filters
)
from keras.utils import (
    to_categorical, # One-hot encoding
    image_dataset_from_directory, # tf.data.Dataset with inferred classes from␣
 ↪folder
    plot_model # Keras model visualization
)
from keras.callbacks import (
    ModelCheckpoint, # Model saving
    EarlyStopping # Early stopping
)
from keras.models import (
    Model, # Keras API model object
    Sequential, # Sequential model object
    load_model # Pre-trained model loading
)
from keras.optimizers.legacy import Adam # Backpropagation

from typing import Tuple, List # Misc. typing (for Classifier)

# Silencing sklearn warnings
import warnings
warnings.filterwarnings("ignore")
warnings.simplefilter("ignore", category = UserWarning)
warnings.simplefilter("ignore", category = FutureWarning)
warnings.simplefilter("ignore", category = DeprecationWarning)
```

1 Reading Data

[ ]:
```
data = pd.read_csv('data.csv')
data.drop(['id', 'Unnamed: 32'], axis = 1, inplace = True) # Dropping id axis␣
 ↪and Unnamed (N/A) axis
sns.countplot(data, x = 'diagnosis') # Plotting output class distribution
```

[ ]:
```
<Axes: xlabel='diagnosis', ylabel='count'>
```



2 Resampling to balance data

[ ]:
```
malignant = data[data['diagnosis'] == 'M'] # Malignant diagnoses
benign = data[data['diagnosis'] == 'B'] # Benign diagnoses
resampledMalignant = resample ( # Data resampling to balance
    malignant,
    replace = True,
    n_samples = len(benign),
    random_state = 42
)

data = pd.concat([benign, resampledMalignant]) # Combining resampled data

features = data.iloc[:,1:] # Independent variables
diagnosis = data.iloc[:,:1].replace({'M': 1, 'B': 0}) # Diagnosis
```

[ ]:
```
features.info()
features.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

```
Index: 714 entries, 19 to 218
Data columns (total 30 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   radius_mean             714 non-null    float64
 1   texture_mean            714 non-null    float64
 2   perimeter_mean          714 non-null    float64
 3   area_mean               714 non-null    float64
 4   smoothness_mean         714 non-null    float64
 5   compactness_mean        714 non-null    float64
 6   concavity_mean          714 non-null    float64
 7   concave points_mean     714 non-null    float64
 8   symmetry_mean           714 non-null    float64
 9   fractal_dimension_mean  714 non-null    float64
 10  radius_se               714 non-null    float64
 11  texture_se              714 non-null    float64
 12  perimeter_se            714 non-null    float64
 13  area_se                 714 non-null    float64
 14  smoothness_se           714 non-null    float64
 15  compactness_se          714 non-null    float64
 16  concavity_se            714 non-null    float64
 17  concave points_se       714 non-null    float64
 18  symmetry_se             714 non-null    float64
 19  fractal_dimension_se    714 non-null    float64
 20  radius_worst            714 non-null    float64
 21  texture_worst           714 non-null    float64
 22  perimeter_worst         714 non-null    float64
 23  area_worst              714 non-null    float64
 24  smoothness_worst        714 non-null    float64
 25  compactness_worst       714 non-null    float64
 26  concavity_worst         714 non-null    float64
 27  concave points_worst    714 non-null    float64
 28  symmetry_worst          714 non-null    float64
 29  fractal_dimension_worst 714 non-null    float64
dtypes: float64(30)
memory usage: 172.9 KB
```

[ ]:

	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	\
19	13.540	14.36	87.46	566.3	0.09779	
20	13.080	15.71	85.63	520.0	0.10750	
21	9.504	12.44	60.34	273.9	0.10240	
37	13.030	18.42	82.61	523.8	0.08983	
46	8.196	16.84	51.71	201.9	0.08600	

	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	\
19	0.08129	0.06664	0.047810	0.1885	
20	0.12700	0.04568	0.031100	0.1967	

```
21          0.06492         0.02956           0.020760        0.1815
37          0.03766         0.02562           0.029230        0.1467
46          0.05943         0.01588           0.005917        0.1769

    fractal_dimension_mean  …  radius_worst  texture_worst  perimeter_worst  \
19                 0.05766  …        15.110          19.26            99.70
20                 0.06811  …        14.500          20.49            96.09
21                 0.06905  …        10.230          15.66            65.13
37                 0.05863  …        13.300          22.81            84.46
46                 0.06503  …         8.964          21.96            57.26

    area_worst  smoothness_worst  compactness_worst  concavity_worst  \
19       711.2           0.14400            0.17730          0.23900
20       630.5           0.13120            0.27760          0.18900
21       314.9           0.13240            0.11480          0.08867
37       545.9           0.09701            0.04619          0.04833
46       242.2           0.12970            0.13570          0.06880

    concave points_worst  symmetry_worst  fractal_dimension_worst
19               0.12880          0.2977                  0.07259
20               0.07283          0.3184                  0.08183
21               0.06227          0.2450                  0.07773
37               0.05013          0.1987                  0.06169
46               0.02564          0.3105                  0.07409

[5 rows x 30 columns]
```

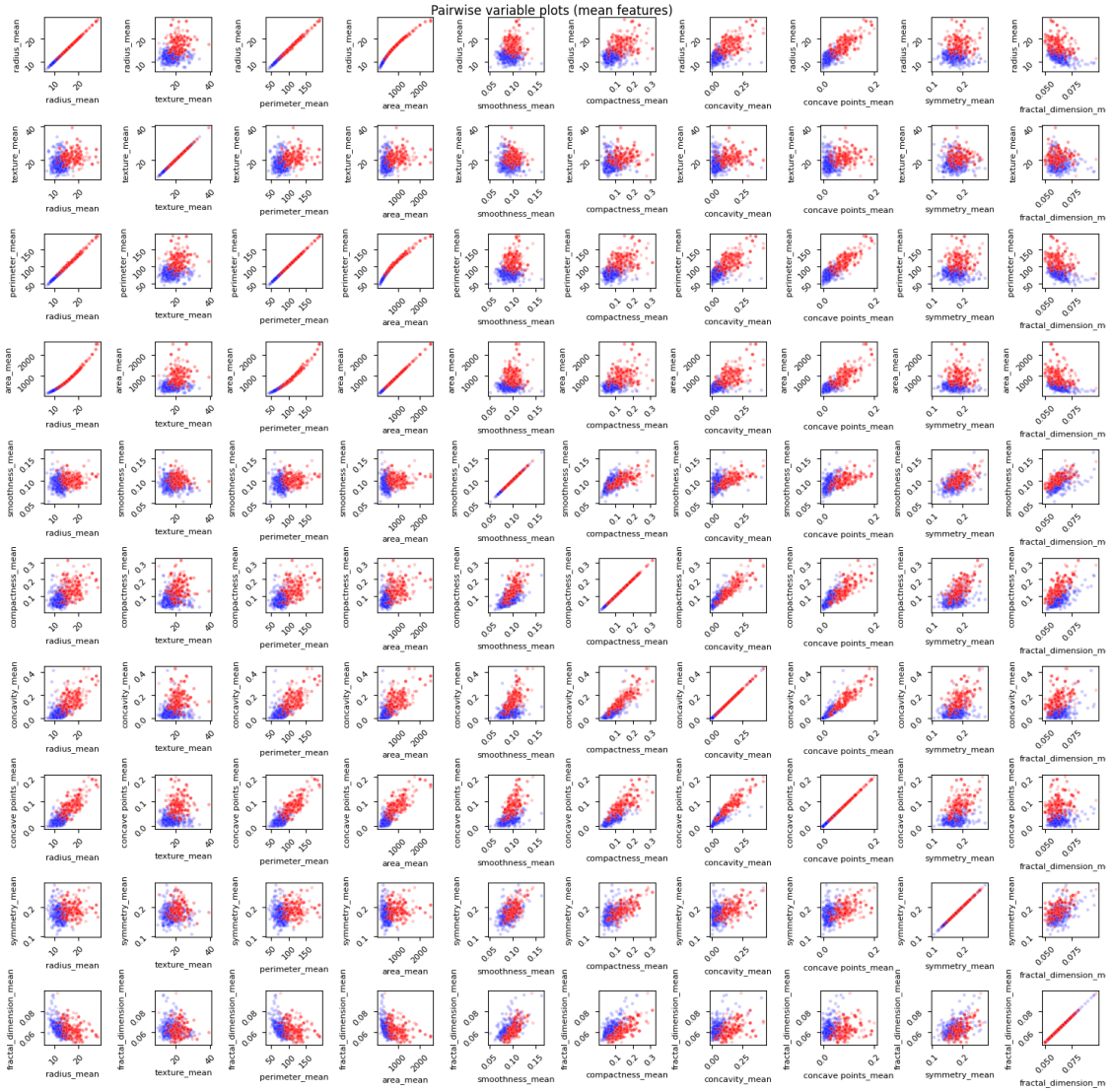
[ ]:
```
np.unique(diagnosis.values)
```

[ ]:
```
array([0, 1])
```

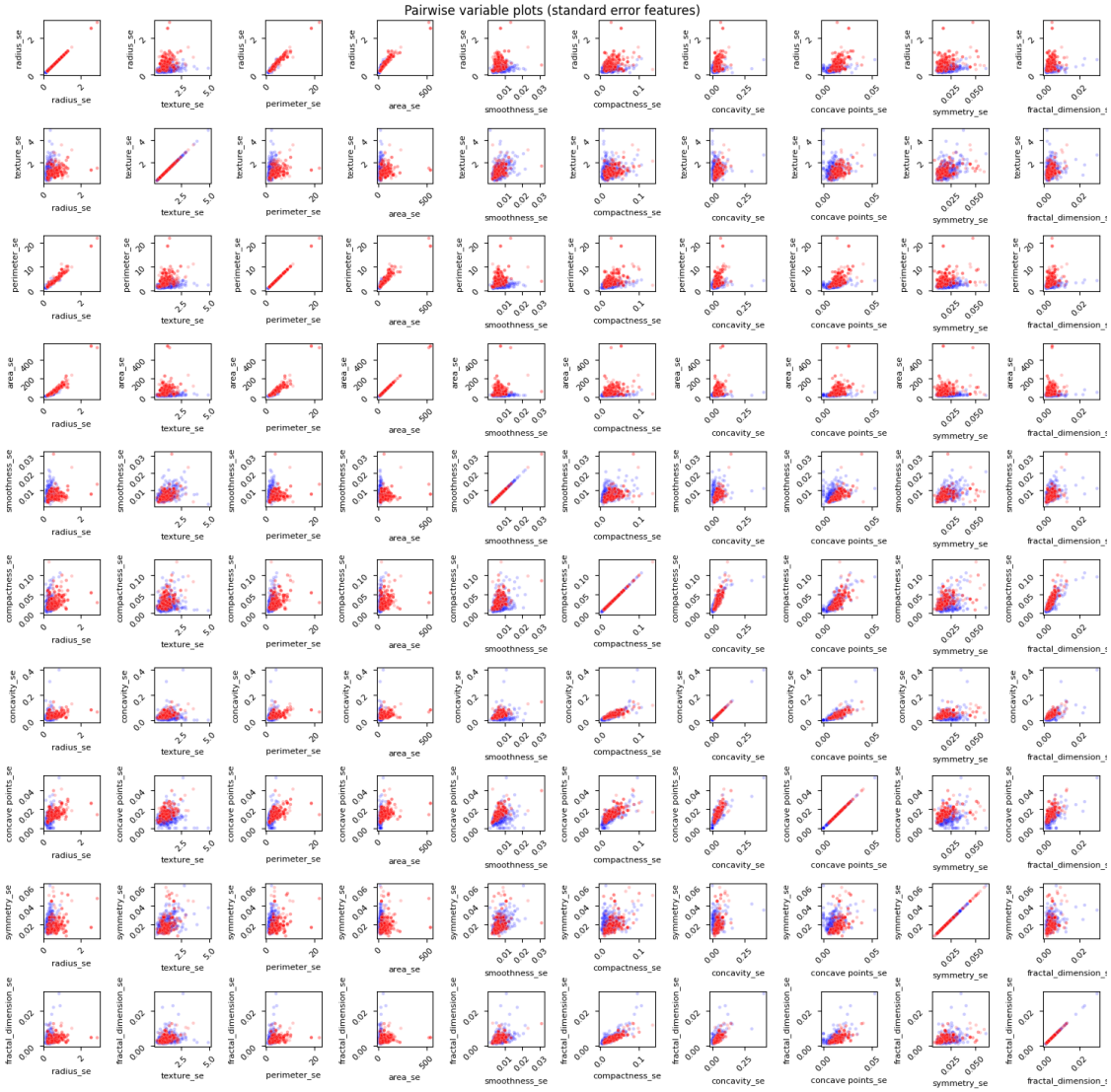
[ ]:
```
featureNames = data.columns.values[1:]
meanFeatures = featureNames[['mean' in feature for feature in featureNames]]
seFeatures = featureNames[['se' in feature for feature in featureNames]]
worstFeatures = featureNames[['worst' in feature for feature in featureNames]]
```

[ ]:
```
def visualiseFeatures(features, name):
    numFeatures = len(features)
    numRows, numCols = 10, 10
    fig, axes = plt.subplots(nrows = numRows, ncols = numCols, figsize = (15,␣
 ↪15))
    fig.suptitle(f'Pairwise variable plots ({name})')
    for i in range(numFeatures):
        for j in range(numFeatures):
            xFeature, yFeature = features[j], features[i]
            ax = axes[i, j]
            sns.scatterplot (
```

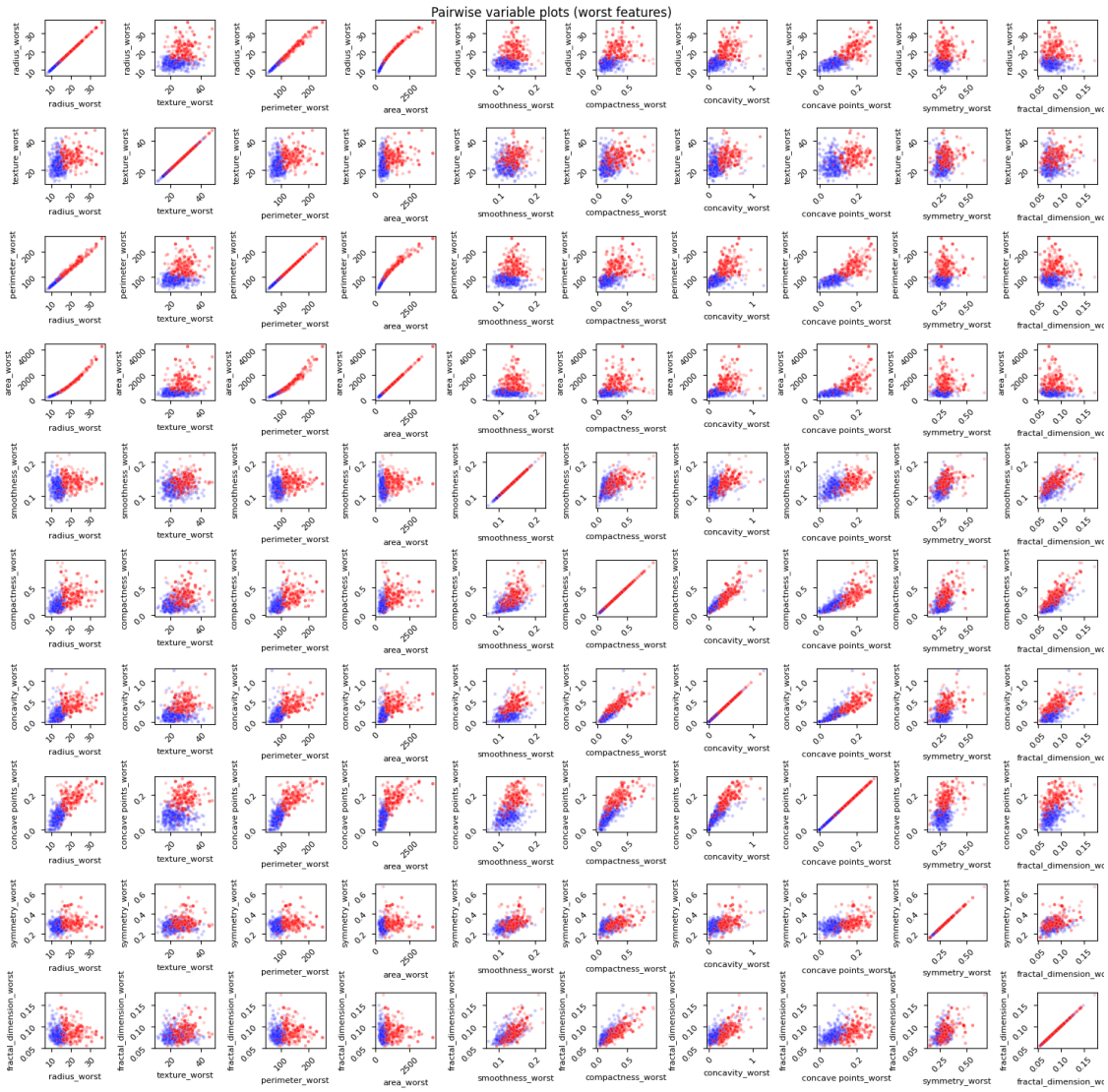
```
                x = data[xFeature], y = data[yFeature],
                c = ['r' if x == 1 else 'b' for x in diagnosis.values],
                alpha = 0.2, s = 10, ax = ax
            )
            ax.set_xlabel(xFeature, fontsize=8)
            ax.set_ylabel(yFeature, fontsize=8)
            ax.tick_params(axis='x', rotation=45, labelsize=8)
            ax.tick_params(axis='y', rotation=45, labelsize=8)

    plt.tight_layout()
    plt.show()
    plt.clf()

visualiseFeatures(meanFeatures, 'mean features')
visualiseFeatures(seFeatures, 'standard error features')
visualiseFeatures(worstFeatures, 'worst features')
```



<Figure size 640x480 with 0 Axes>



<Figure size 640x480 with 0 Axes>



<Figure size 640x480 with 0 Axes>

3 Data splitting

No data reserved for validation since classical machine learning does not require early stopping for training.

[ ]:
```
xTrain, xTest, yTrain, yTest = train_test_split ( # Shuffling and splitting␣
 ↪data into train and test
    features,
    diagnosis,
    test_size = 0.15,
    stratify = diagnosis, # Balanced diagnoses in training and testing data
```

```
    random_state = 42
)

#yTrain, yTest = yTrain.values.ravel(), yTest.values.ravel() # Flattening output
```

[ ]:
```
# Evaluation of a model by visualizing the confusion matrix and displaying the␣
 ↪f1 score
def evaluate(model, modelName):
    # Set subplots (1 by 2)
    fig, (ax1, ax2) = plt.subplots (
        1, 2,
        figsize = [12, 4],
        dpi = 300,
        clear = True
    )
    # Predict and score data
    trainingPrediction = model.predict(xTrain)
    testingPrediction = model.predict(xTest)
    interpolationF1 = f1_score(yTrain, trainingPrediction)
    extrapolationF1 = f1_score(yTest, testingPrediction)
    # Overall title
    fig.suptitle(f'{modelName} Confusion Matrices')
    # Subplot titles
    ax1.title.set_text(('Interpolation (f1 = %.4f)' % (interpolationF1)))
    ax2.title.set_text(('Extrapolation (f1 = %.4f)' % (extrapolationF1)))
    # Confusion matrices
    interpolationConfusion = confusion_matrix(yTrain, trainingPrediction)
    extrapolationConfusion = confusion_matrix(yTest, testingPrediction)
    # Visualizing confusion matrices
    sns.heatmap (
        interpolationConfusion, annot = True, fmt = 'd',
        cmap = 'YlGnBu', ax = ax1, square = True,
        xticklabels = ['Benign', 'Malignant'],
        yticklabels = ['Benign', 'Malignant']
    )
    # Label heatmap axes
    ax1.set(xlabel = "True Class", ylabel = "Predicted Class")
    sns.heatmap (
        extrapolationConfusion, annot = True, fmt = 'd',
        cmap = 'YlGnBu', ax = ax2, square = True,
        xticklabels = ['Benign', 'Malignant'],
        yticklabels = ['Benign', 'Malignant']
    )
    # Label heatmap axes
    ax2.set(xlabel = "True Class", ylabel = "Predicted Class")
    plt.show()
    return extrapolationF1
```

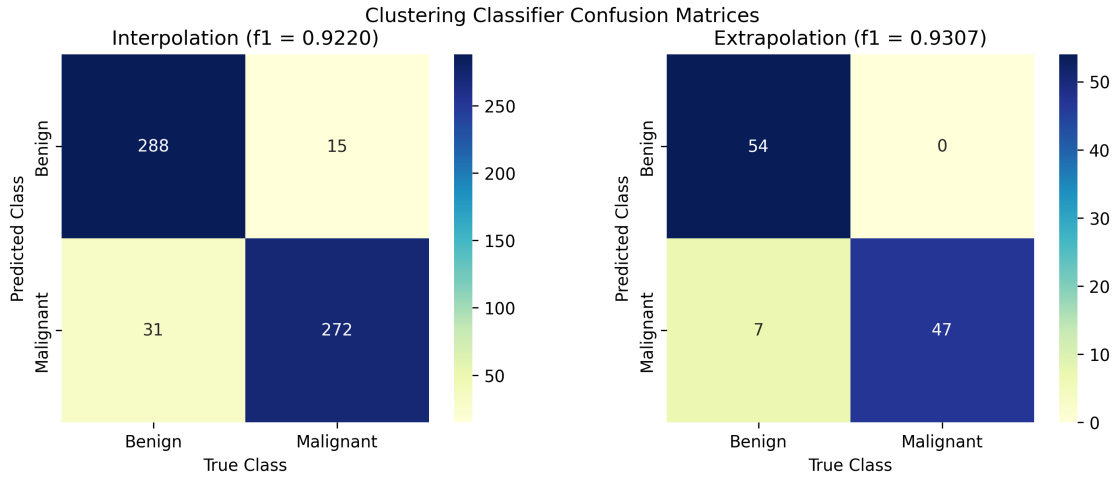
4 Model 1: Nearest Neighbors Algorithm (clustering)

[ ]:
```
# Defining hyperparameters to be searched
clusteringParams = {
    'n_neighbors': np.arange(5, 55, 5), # (5, 10, ..., 50)
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
}

# Cross-validation of 15 different hyperparameter combinations
bestClusteringParams = RandomizedSearchCV (
    KNC(),
    clusteringParams,
    scoring = 'f1',
    n_iter = 15,
    random_state = 42
).fit(xTrain, yTrain).best_params_
```

[ ]:
```
print(bestClusteringParams)
```

```
{'n_neighbors': 25, 'algorithm': 'auto'}
```

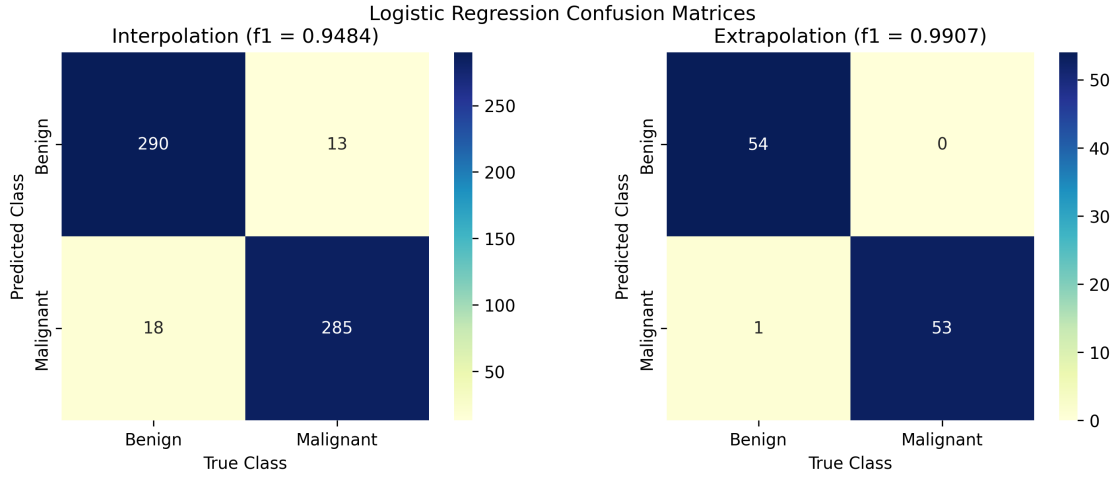
[ ]:
```
# Initializing the best found model
clusteringClassifier = KNC(**bestClusteringParams)

clusteringClassifier.fit(xTrain, yTrain)

# Plotting the confusion matrices and displaying the f1 score for interpolation␣
 ↪and extrapolation
clusteringF1 = evaluate(clusteringClassifier, 'Clustering Classifier')
```



5 Model 2: Logistic Regression

[ ]:
```
# Repeat the search steps for each model
logisticParams = {
    'C': np.linspace(0.5, 4.5, 20)
}

bestLogisticParams = RandomizedSearchCV (
    LR(),
    logisticParams,
    scoring = 'f1',
    n_iter = 1000,
    random_state = 42
).fit(xTrain, yTrain).best_params_

print(bestLogisticParams)

logisticClassifier = LR(**bestLogisticParams)

logisticClassifier.fit(xTrain, yTrain)

logisticF1 = evaluate(logisticClassifier, 'Logistic Regression')
```

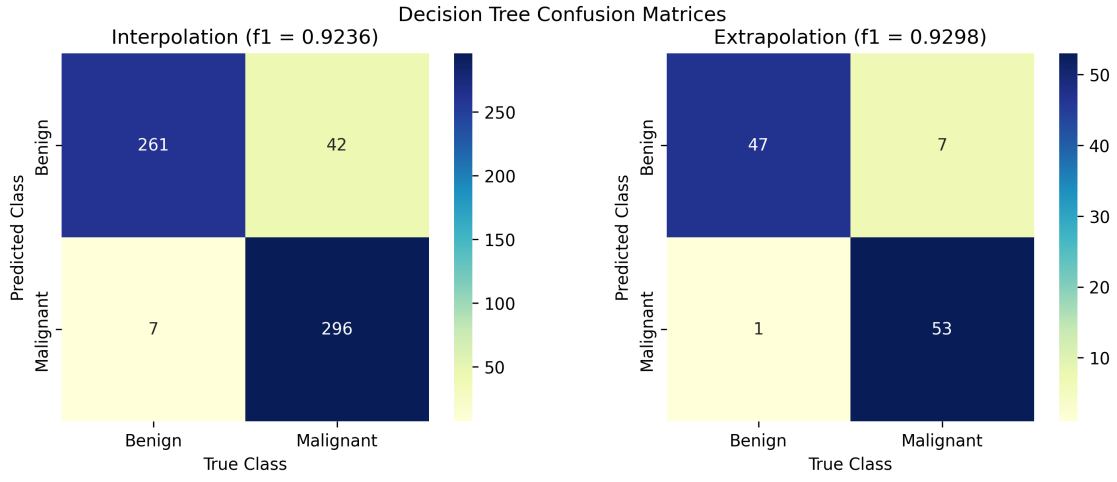
```
{'C': 3.4473684210526314}
```



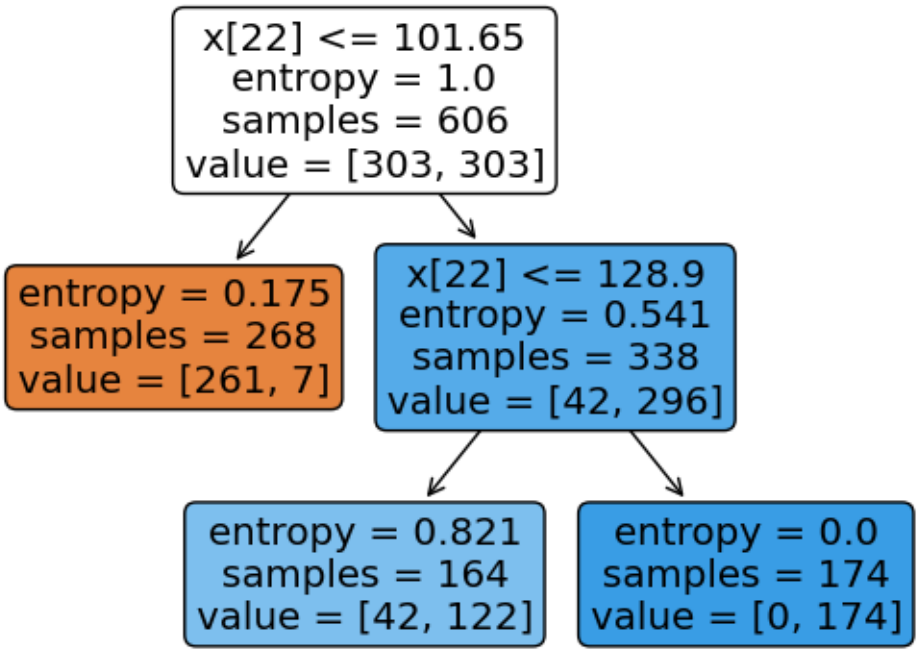
6 Model 3: Decision Tree Classifier

[ ]:
```
# Repeat the search steps for each model
decisionTreeParams = {
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': np.append(np.arange(2, 21), None),
    'min_samples_leaf': np.linspace(0.01, 0.3, 50),
    'random_state': [42]
}

bestDecisionTreeParams = RandomizedSearchCV (
    DTC(),
    decisionTreeParams,
    scoring = 'f1',
    n_iter = 15,
    random_state = 42
).fit(xTrain, yTrain).best_params_

print(bestDecisionTreeParams)

decisionTree = DTC(**bestDecisionTreeParams)

decisionTree.fit(xTrain, yTrain)

decisionTreeF1 = evaluate(decisionTree, 'Decision Tree')

# Visualizing the decision tree
plot_tree(decisionTree, filled = True, rounded = True)
```

```
{'random_state': 42, 'min_samples_leaf': 0.27040816326530615, 'max_depth': 7,
'criterion': 'entropy'}
```



[ ]:
```
[Text(0.4, 0.8333333333333334, 'x[22] <= 101.65\nentropy = 1.0\nsamples =
606\nvalue = [303, 303]'),
 Text(0.2, 0.5, 'entropy = 0.175\nsamples = 268\nvalue = [261, 7]'),
 Text(0.6, 0.5, 'x[22] <= 128.9\nentropy = 0.541\nsamples = 338\nvalue = [42,
296]'),
 Text(0.4, 0.16666666666666666, 'entropy = 0.821\nsamples = 164\nvalue = [42,
122]'),
 Text(0.8, 0.16666666666666666, 'entropy = 0.0\nsamples = 174\nvalue = [0,
174]')]
```

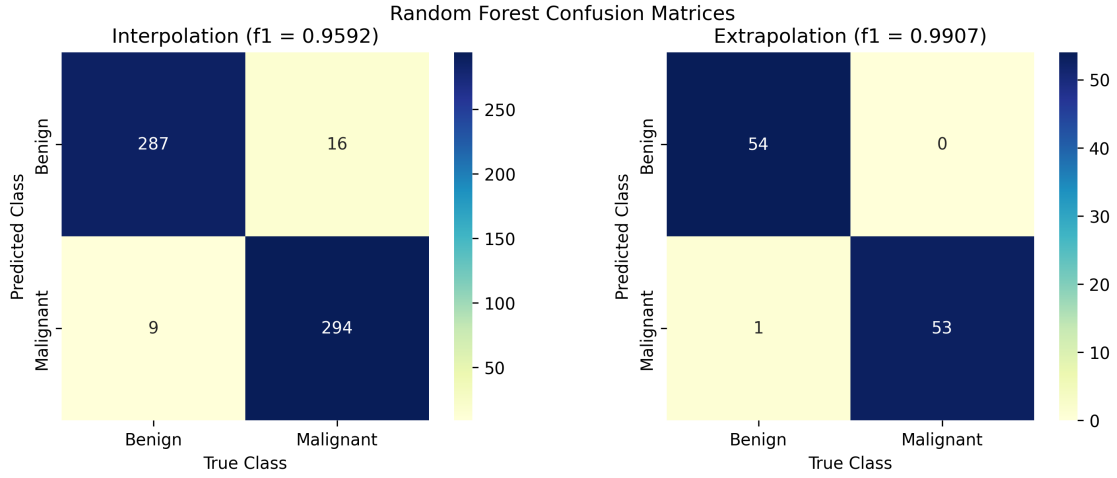


7 Model 4: Random Forest Classifier

[ ]:
```
# Repeat the search steps for each model
randomForestParams = {
    'n_estimators': np.arange(50, 300, 5),
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': np.append(np.arange(2, 21), None),
    'min_samples_leaf': np.linspace(0.01, 0.3, 50),
    'random_state': [42]
}

bestRandomForestParams = RandomizedSearchCV (
```

```
    RFC(),
    randomForestParams,
    scoring = 'f1',
    n_iter = 15,
    random_state = 42
).fit(xTrain, yTrain).best_params_

print(bestRandomForestParams)

randomForest = RFC(**bestRandomForestParams)

randomForest.fit(xTrain, yTrain)

randomForestF1 = evaluate(randomForest, 'Random Forest')
```

```
{'random_state': 42, 'n_estimators': 185, 'min_samples_leaf':
0.0336734693877551, 'max_depth': 7, 'criterion': 'log_loss'}
```

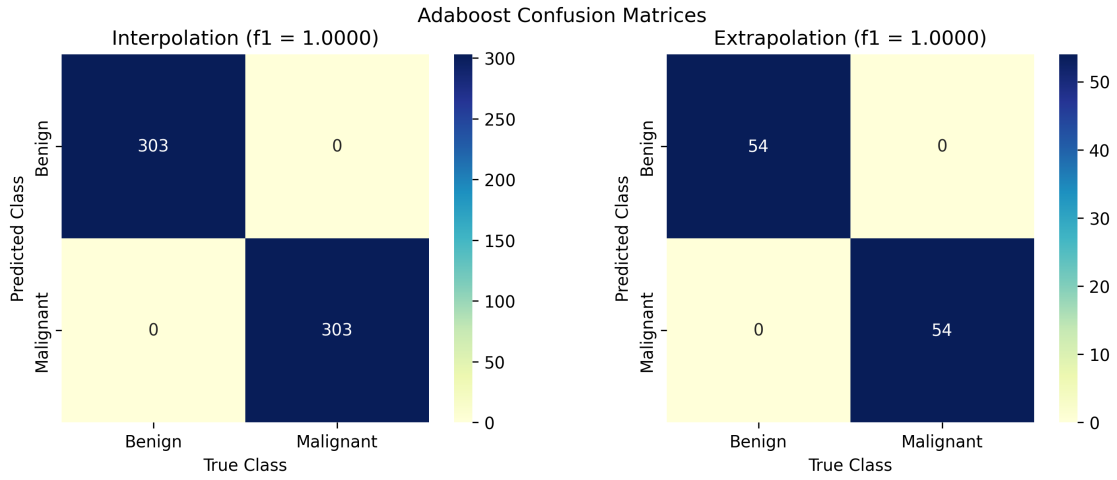


8 Model 5: AdaBoost Classifier

[ ]:
```
# Repeat the search steps for each model
adaboostParams = {
    'n_estimators': np.arange(10, 100, 5),
    'learning_rate': np.linspace(0.4, 1.0, 20),
    'random_state': [42]
}

bestAdaboostParams = RandomizedSearchCV (
    ABC(),
    adaboostParams,
```

```
    scoring = 'f1',
    n_iter = 15,
    random_state = 42
).fit(xTrain, yTrain).best_params_

print(bestAdaboostParams)

adaboost = ABC(**bestAdaboostParams)

adaboost.fit(xTrain, yTrain)

adaboostF1 = evaluate(adaboost, 'Adaboost')
```

```
{'random_state': 42, 'n_estimators': 80, 'learning_rate': 0.8105263157894738}
```

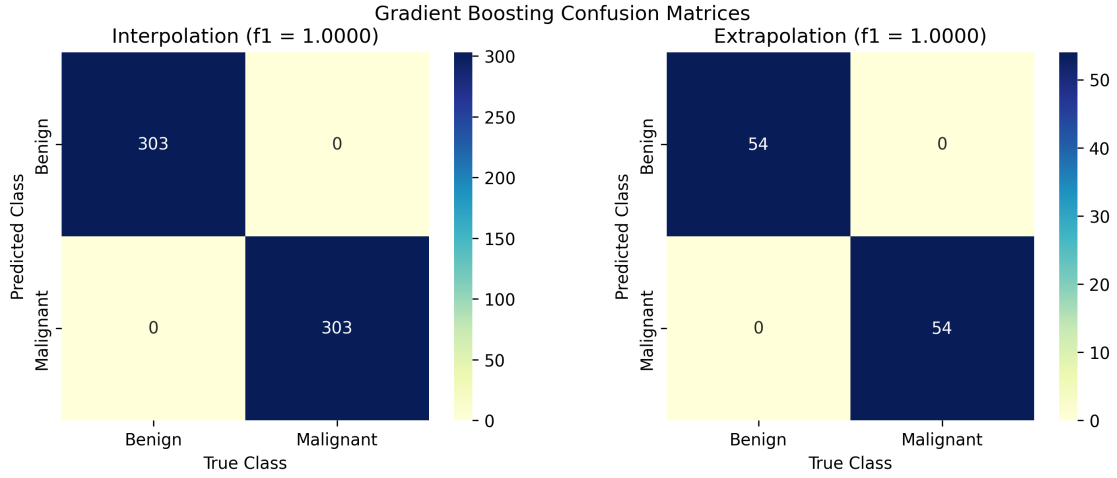


9 Model 6: Gradient Boosting Classifier

[ ]:
```
# Repeat the search steps for each model
gradientBoostingParams = {
    'loss': ['log_loss', 'exponential'],
    'n_estimators': np.arange(50, 200, 5),
    'learning_rate': np.linspace(0.4, 1.0, 20),
    'min_samples_leaf': np.linspace(0.01, 0.3, 50),
    'max_depth': [2, 3, 4, 5],
    'random_state': [42]
}

bestGradientBoostingParams = RandomizedSearchCV (
    GBC(),
```

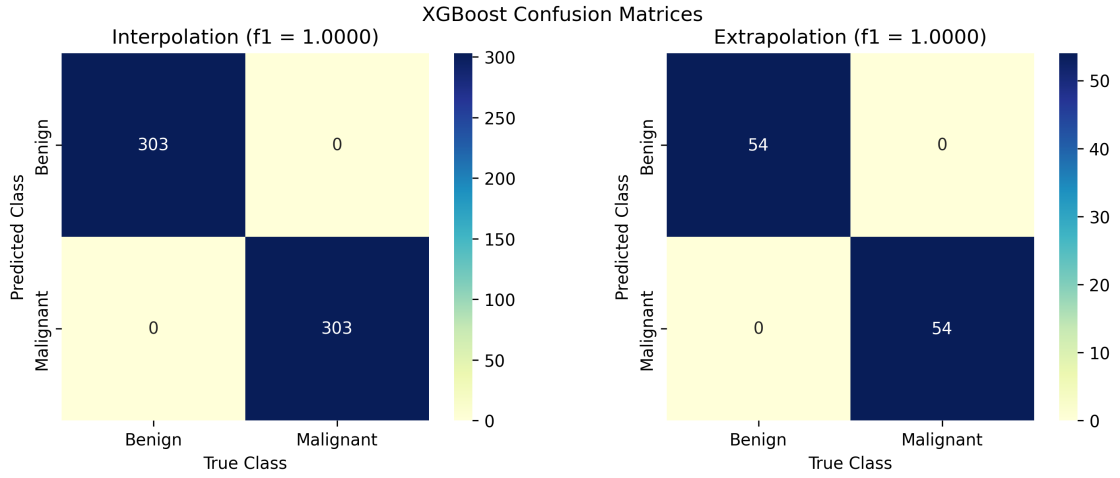
```
    gradientBoostingParams,
    scoring = 'f1',
    n_iter = 15,
    random_state = 42
).fit(xTrain, yTrain).best_params_

print(bestGradientBoostingParams)

gradient = GBC(**bestGradientBoostingParams)

gradient.fit(xTrain, yTrain)

gradientF1 = evaluate(gradient, 'Gradient Boosting')
```

```
{'random_state': 42, 'n_estimators': 135, 'min_samples_leaf':
0.2763265306122449, 'max_depth': 3, 'loss': 'log_loss', 'learning_rate':
0.7789473684210526}
```



10 Model 7: XGBoost Classifier

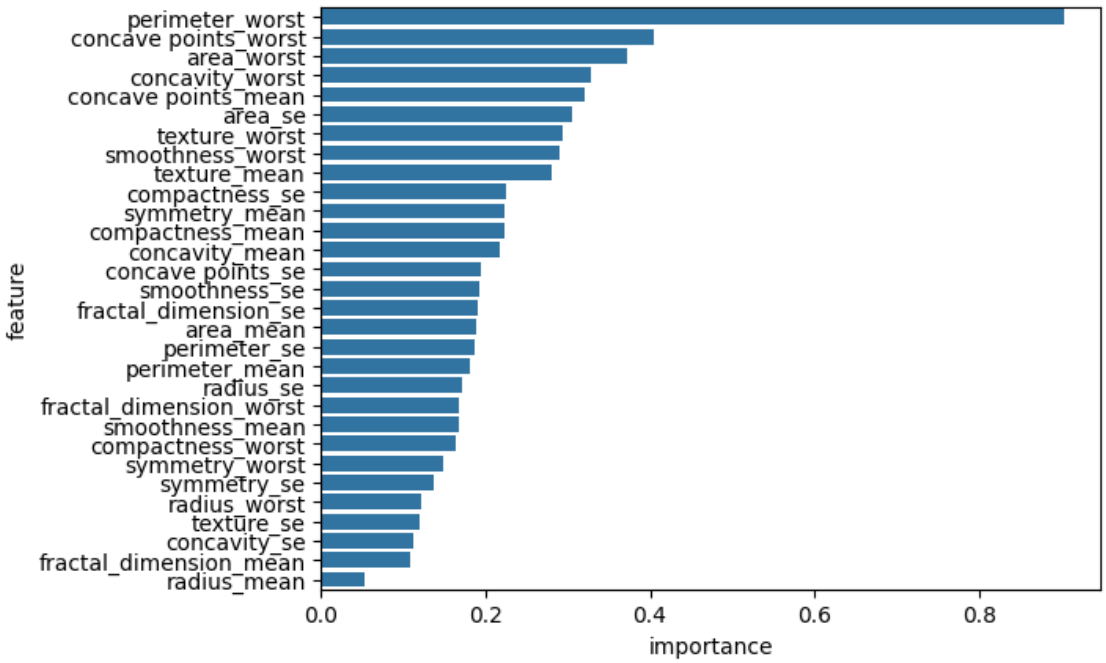
[ ]:
```
xgboost = XBC(num_parallel_tree = 185) # Best n_estimators from Random Forest

xgboost.fit(xTrain, yTrain)

xgboostF1 = evaluate(xgboost, 'XGBoost')
```



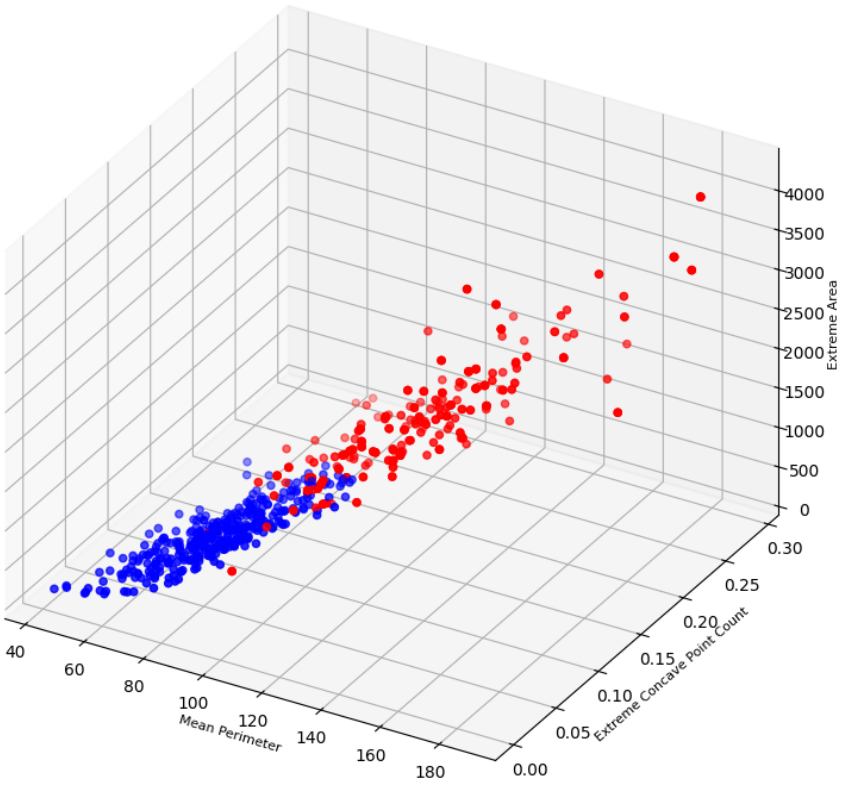
11 Summarizing model results

[ ]:
```
# DataFrame summarizing trained model information and results
scores = pd.DataFrame(data = {
    'model name': ['clustering', 'logistic', 'decision tree',
                   'random forest', 'adaboost', 'gradient boosting',
                   'xgboost'],
    'model': [clusteringClassifier, logisticClassifier, decisionTree,
              randomForest, adaboost, gradient, xgboost],
    'f1 score': [clusteringF1, logisticF1, decisionTreeF1,
                 randomForestF1, adaboostF1, gradientF1,
                 xgboostF1],
    'importance': [None, logisticClassifier.coef_[0], decisionTree.
 ↪feature_importances_,
                   randomForest.feature_importances_, adaboost.
 ↪feature_importances_,
                   gradient.feature_importances_, xgboost.feature_importances_]
})
scores.set_index('model name', inplace = True)

# Estimating probabilities
scores['proba'] = scores['model'].apply(lambda model: model.
 ↪predict_proba(xTest))
# Finding loss scores
scores['loss'] = scores['proba'].apply(lambda proba: log_loss(yTest, proba))

# Sorting by descending f1 score then ascending loss
scores.sort_values(by = ['f1 score', 'loss'], ascending = [False, True],␣
 ↪inplace = True)
```

```
featureNames = data.columns.tolist()[1:]
x = scores['importance']

# Linear mapping of data to [0, 1]
def normalize(x):
    if x is None:
        return None
    x = np.array(x)
    return (x - min(x)) / (max(x) - min(x))

# Normalizing scores (shared scale of [0, 1] across all models)
scores['normalizedImportance'] = scores['importance'].apply(normalize)

# Extracting average importance values
importances = np.array([x for x in scores['normalizedImportance'].tolist() if x␣
 ↪is not None])
averageImportances = [np.mean(x) for x in np.transpose(importances)]
importanceFrame = pd.DataFrame(data = {
    'feature': featureNames,
    'importance': averageImportances
})
# Sorting and bar-plotting average importance values
importanceFrame.sort_values(by = 'importance', inplace = True, ascending =␣
 ↪False)
sns.barplot(x = importanceFrame['importance'], y = importanceFrame['feature'])
plt.show()
```

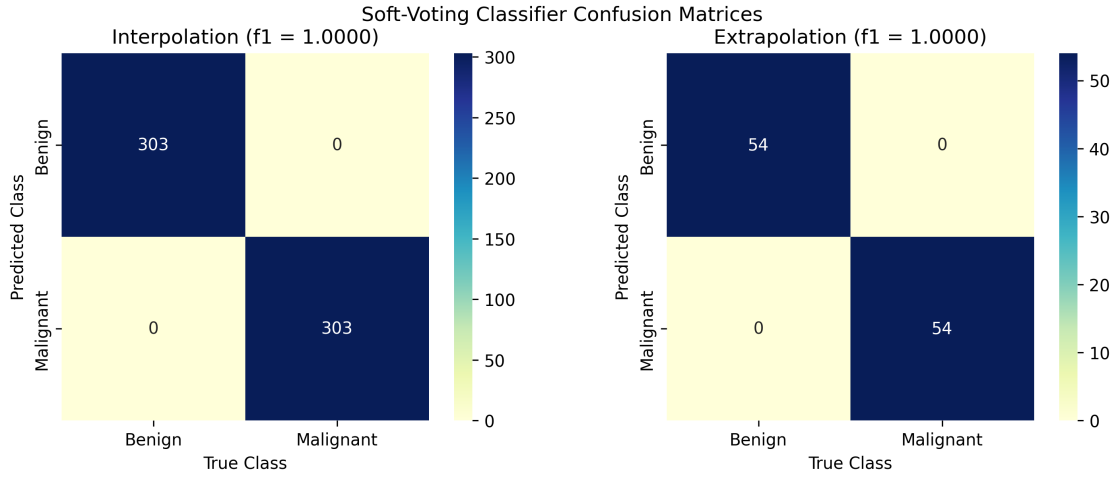


[ ]:
```
fig = plt.figure(figsize = (16, 12))

ax = fig.add_subplot(111, projection = '3d', proj_type = 'ortho')

ax.scatter (
    data['perimeter_mean'],
    data['concave points_worst'],
    data['area_worst'],
    c = ['r' if x == 1 else 'b' for x in diagnosis.values]
)

ax.set_box_aspect(aspect = None, zoom = 0.8)

ax.set_xlabel('Mean Perimeter', loc = 'left', fontsize = 8)
ax.set_ylabel('Extreme Concave Point Count', loc = 'bottom', fontsize = 8)
ax.set_zlabel('Extreme Area', fontsize = 8)


plt.show()
```



12 Model 8: Soft-Voting Classifier (ensemble)

```
[ ]: modelsUsed = scores[np.logical_and(scores['f1 score'] > 0.95, scores['loss'] <␣
     ↪0.15)]['model']
     voter = VC (
         estimators = list(zip(scores.index, modelsUsed.tolist())),
         voting = 'soft'
     )

     voter.fit(xTrain, yTrain)

     voterF1 = evaluate(voter, 'Soft-Voting Classifier')
```



[ ]:
```
voterProba = voter.predict_proba(xTest)
log_loss(yTest, voterProba)
```

[ ]: 0.039368820733836664

13 Model 9: Feedforward Neural Network

[ ]:
```
class FNNClassifier:
    def __init__(
        self,
        inputShape,
        outputClassCount: int,
        denseLayerCount: int,
        denseUnits: List[int],
        denseDropoutRates: List[float],
        trainingSize: float = 0.6,
        testingSize: float = 0.2,
        validationSize: float = 0.2,
        srand: int = 42
    ):
        self.inputShape = inputShape
        self.outputClassCount = outputClassCount
        self.denseLayerCount = denseLayerCount
        self.denseUnits = denseUnits
        self.denseDropoutRates = denseDropoutRates
        self.srand = srand
        self.layers = []

        assert self.denseLayerCount > 0
        assert len(self.denseUnits) == self.denseLayerCount
```

```
        assert len(self.denseDropoutRates) == self.denseLayerCount

        self._construct()

    def _construct(self) -> None:
        self.layers.append(Input(shape = self.inputShape, name = "Input"))
        for denseLayer in range(self.denseLayerCount):
            self.layers.append (
                Dropout (
                    rate = self.denseDropoutRates[denseLayer],
                    name = f"Dropout_{denseLayer + 1}"
                ) (
                    BatchNormalization (
                        name = f"Batch_Normalization_{denseLayer + 1}"
                    ) (
                        LeakyReLU (
                            name = f"Leaky_ReLU_{denseLayer + 1}"
                        ) (
                            Dense (
                                units = self.denseUnits[denseLayer],
                                name = f"Dense_{denseLayer + 1}"
                            ) (
                                self.layers[-1]
                            )
                        )
                    )
                )
            )
        self.layers.append (
            Softmax (name = "Softmax") (
                self.layers[-1]
            )
        )
        self.input = self.layers[0]
        self.output = self.layers[-1]
        self.classifier = Model (
            inputs = [self.input],
            outputs =[self.output]
        )
        return None
    pass

fnnclassifier = FNNClassifier (
    inputShape = (30,),
    outputClassCount = 2,
    denseLayerCount = 8,
    denseUnits = [60, 50, 40, 30, 20, 15, 5, 2],
```

```
    denseDropoutRates = [0.3, 0.4, 0.35, 0.2, 0.2, 0.2, 0, 0]
)

# The below code was used to train the classifier.
# It has been replaced by code loading a pretrained model saved by the below␣
 ↪code.
"""
fnnmodel = classifier.classifier

fnnmodel.compile(optimizer = Adam(0.0005), loss = 'categorical_crossentropy',␣
 ↪metrics = ['accuracy'])

fnnmodel.fit (
    x = xTrain,
    y = to_categorical(yTrain, 2),
    epochs = 2000,
    callbacks = [
        ModelCheckpoint (
            "/Users/ericssonlin/code/Python/Vogelsberger/final/nn.keras",
            monitor = 'val_loss',
            save_best_only = True,
            save_weights_only = False,
            mode = 'min',
            verbose = 1
        )
    ],
    validation_data = (xTest, to_categorical(yTest, 2))
)
"""

fnnclassifier.classifier = load_model('nn.keras')

fnnmodel = fnnclassifier.classifier
```

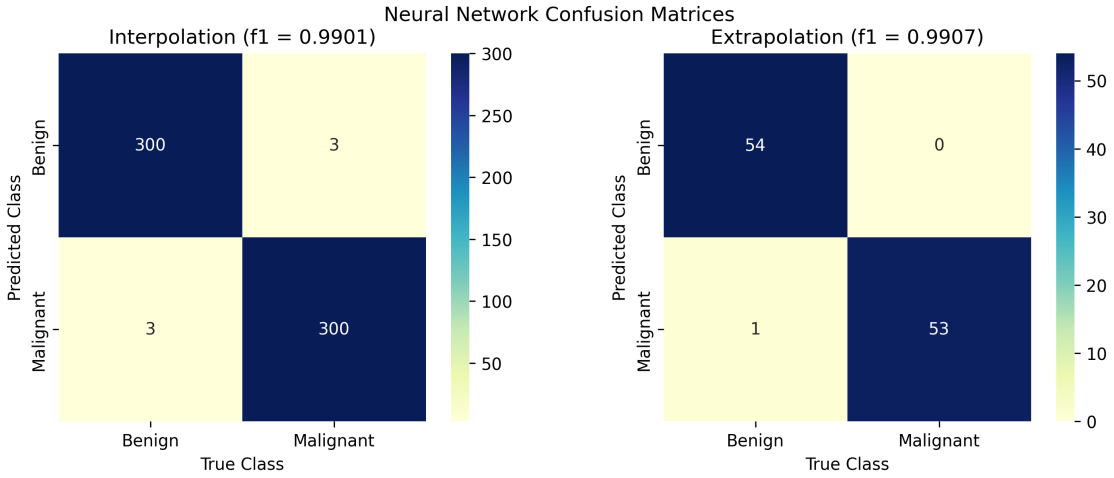
```
2024-03-16 19:22:52.910102: I metal_plugin/src/device/metal_device.cc:1154]
Metal device set to: Apple M1
2024-03-16 19:22:52.910129: I metal_plugin/src/device/metal_device.cc:296]
systemMemory: 16.00 GB
2024-03-16 19:22:52.910139: I metal_plugin/src/device/metal_device.cc:313]
maxCacheSize: 5.33 GB
2024-03-16 19:22:52.910183: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:306]
Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel
may not have been built with NUMA support.
2024-03-16 19:22:52.910203: I
tensorflow/core/common_runtime/pluggable_device/pluggable_device_factory.cc:272]
Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0
```

```
MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id:
<undefined>)
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.
```

[ ]:
```
# Set subplots (1 by 2)
fig, (ax1, ax2) = plt.subplots (
    1, 2,
    figsize = [12, 4],
    dpi = 300,
    clear = True
)
# Predict and score data
fnnTrainingPrediction = np.argmax(fnnmodel.predict(xTrain), axis = 1)
fnnTestingPrediction = np.argmax(fnnmodel.predict(xTest), axis = 1)
# Overall title
fig.suptitle(f'Neural Network Confusion Matrices')
# Subplot titles
ax1.title.set_text(('Interpolation (f1 = %.4f)' % (f1_score(yTrain,␣
 ↪fnnTrainingPrediction))))
ax2.title.set_text(('Extrapolation (f1 = %.4f)' % (f1_score(yTest,␣
 ↪fnnTestingPrediction))))
# Visualizing confusion matrices
sns.heatmap (
    confusion_matrix(yTrain, fnnTrainingPrediction), # Matrix
    annot = True, # Label values
    fmt = 'd', # Integer output
    cmap = 'YlGnBu', # Colormap
    ax = ax1, # Subplot axis
    square = True, # Equal aspect
    xticklabels = ['Benign', 'Malignant'], # Ticklabels
    yticklabels = ['Benign', 'Malignant']
)
# Label heatmap axes
ax1.set(xlabel = "True Class", ylabel = "Predicted Class")
sns.heatmap (
    confusion_matrix(yTest, fnnTestingPrediction), # Matrix
    annot = True, # Label values
    fmt = 'd', # Integer output
    cmap = 'YlGnBu', # Colormap
    ax = ax2, # Subplot axis
    square = True, # Equal aspect
    xticklabels = ['Benign', 'Malignant'], # Ticklabels
    yticklabels = ['Benign', 'Malignant']
)
# Label heatmap axes
```

```
ax2.set(xlabel = "True Class", ylabel = "Predicted Class")
plt.show()
```

```
2024-03-16 19:22:54.087333: I
tensorflow/core/grappler/optimizers/custom_graph_optimizer_registry.cc:117]
Plugin optimizer for device_type GPU is enabled.

19/19 [==============================] - 1s 15ms/step
4/4 [==============================] - 0s 45ms/step
```



[ ]:
```
fnnDF = pd.DataFrame(data = {
    'model name': ['feedforward neural network'],
    'model': [fnnmodel],
    'f1 score': [f1_score(yTest, fnnTestingPrediction)],
    'importance': [None]
}).set_index('model name')

fnnDF['proba'] = fnnDF['model'].apply(lambda model: model.predict(xTest))
fnnDF['loss'] = fnnDF['proba'].apply(lambda proba: log_loss(yTest, proba))

scores = pd.concat([scores, fnnDF])
scores.sort_values(by = ['f1 score', 'loss'], ascending = [False, True],␣
 ↪inplace = True)
scores
```

```
4/4 [==============================] - 0s 10ms/step
```

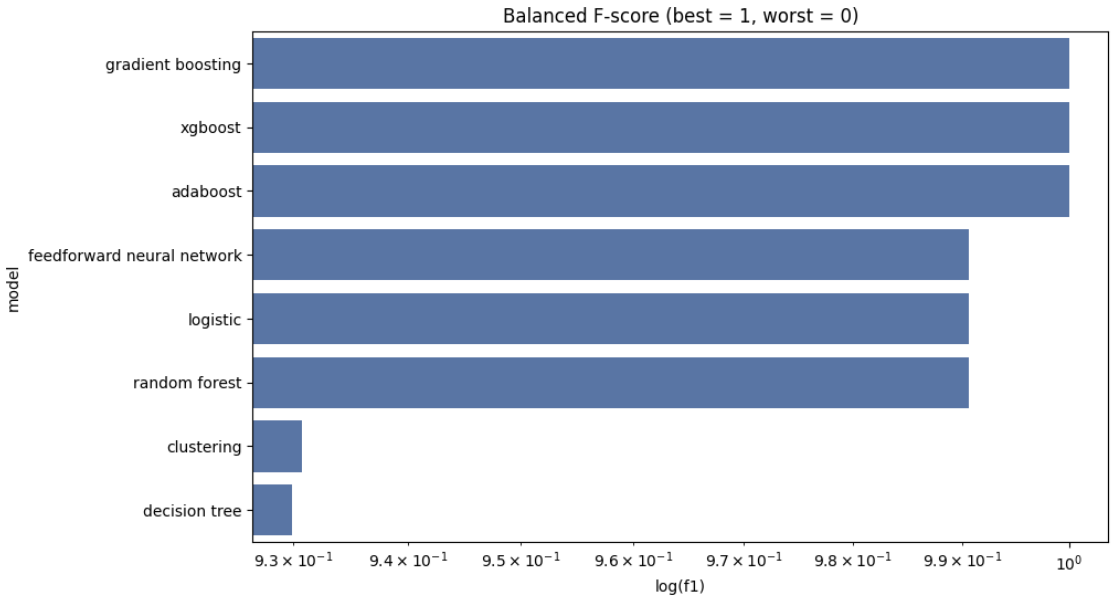
[ ]:
```
                                                                model  \
model name
gradient boosting          ([DecisionTreeRegressor(criterion='friedman_ms…
xgboost                    XGBClassifier(base_score=None, booster=None, c…
```

```
adaboost                     (DecisionTreeClassifier(max_depth=1, random_st…
feedforward neural network   <keras.src.engine.functional.Functional object…
logistic                             LogisticRegression(C=3.4473684210526314)
random forest                (DecisionTreeClassifier(criterion='log_loss', …
clustering                                KNeighborsClassifier(n_neighbors=25)
decision tree                DecisionTreeClassifier(criterion='entropy', ma…

                             f1 score  \
model name
gradient boosting            1.000000
xgboost                      1.000000
adaboost                     1.000000
feedforward neural network   0.990654
logistic                     0.990654
random forest                0.990654
clustering                   0.930693
decision tree                0.929825

                                                                   importance  \
model name
gradient boosting            [0.0, 0.017142309314311648, 0.0, 1.23791389553…
xgboost                      [0.009822933, 0.013481897, 0.045089897, 0.0259…
adaboost                     [0.0, 0.0625, 0.0125, 0.0125, 0.025, 0.0375, 0…
feedforward neural network                                                None
logistic                     [-1.835631337690813, 0.36569104131138463, -0.1…
random forest                [0.046675205608290915, 0.008456193922655283, 0…
clustering                                                                None
decision tree                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …

                                                                     proba  \
model name
gradient boosting            [[7.782663402622347e-13, 0.9999999999992217], …
xgboost                      [[0.00026518106, 0.9997348], [0.9995727, 0.000…
adaboost                     [[0.3745495988241276, 0.6254504011758725], [0…
feedforward neural network   [[7.36489e-05, 0.9999263], [0.9994062, 0.00059…
logistic                     [[4.884981308350689e-15, 0.9999999999999951], …
random forest                [[0.0004506194750097189, 0.9995493805249903], …
clustering                   [[0.0, 1.0], [1.0, 0.0], [1.0, 0.0], [1.0, 0.0…
decision tree                [[0.0, 1.0], [0.9738805970149254, 0.0261194029…

                                 loss  \
model name
gradient boosting            0.001141
xgboost                      0.008081
adaboost                     0.485164
feedforward neural network   0.025353
logistic                     0.060986
```

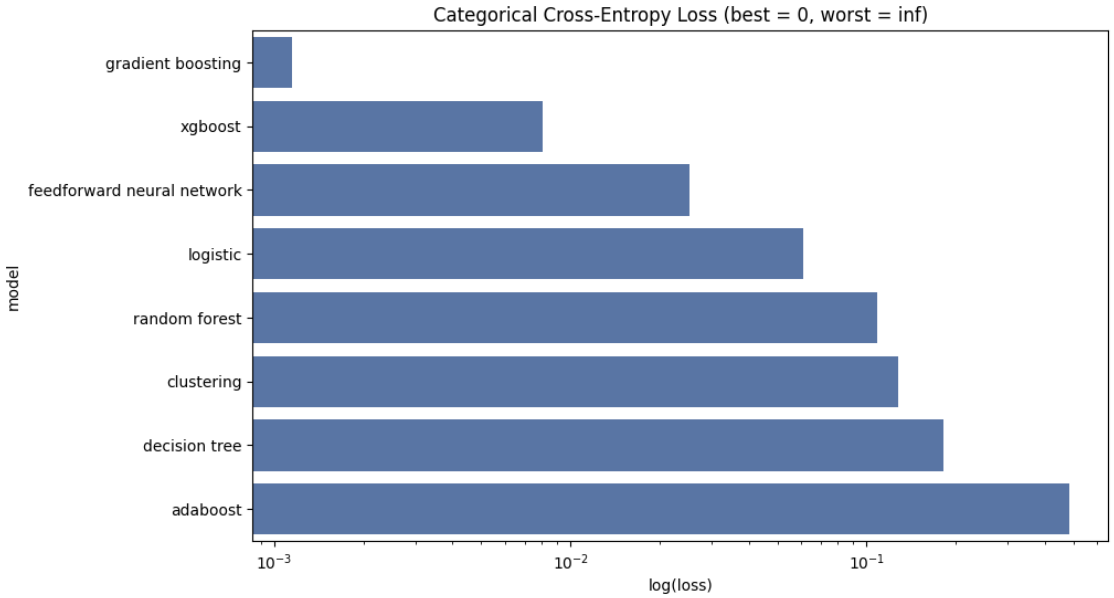
```
random forest                0.108832
clustering                   0.128549
decision tree                0.182867
```

```
                                                    normalizedImportance
model name
gradient boosting            [0.0, 0.02303338200260578, 0.0, 1.663331533655…
xgboost                      [0.013517046, 0.018765643, 0.064105704, 0.0366…
adaboost                     [0.0, 0.8333333333333334, 0.16666666666666669,…
feedforward neural network                                             NaN
logistic                     [0.03531801971569794, 0.7590620678318746, 0.58…
random forest                [0.2693816845977932, 0.04396384244517416, 0.26…
clustering                                                            None
decision tree                [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, …
```

[ ]:
```
sns.set_palette('deep')
plt.figure(figsize = (10, 6))
sns.barplot(data = scores, x = 'f1 score', y = 'model name')
plt.title('Balanced F-score (best = 1, worst = 0)')
plt.xscale('log'); plt.xlabel('log(f1)'); plt.ylabel('model')
plt.show()
plt.clf()

plt.figure(figsize = (10, 6))
sns.barplot(data = scores.sort_values(by = ['loss', 'f1 score'], ascending =␣
 ↪True),
            x = 'loss', y = 'model name')
plt.title('Categorical Cross-Entropy Loss (best = 0, worst = inf)')
plt.xscale('log'); plt.xlabel('log(loss)'); plt.ylabel('model')
plt.show()
plt.clf()
```



<Figure size 640x480 with 0 Axes>



<Figure size 640x480 with 0 Axes>

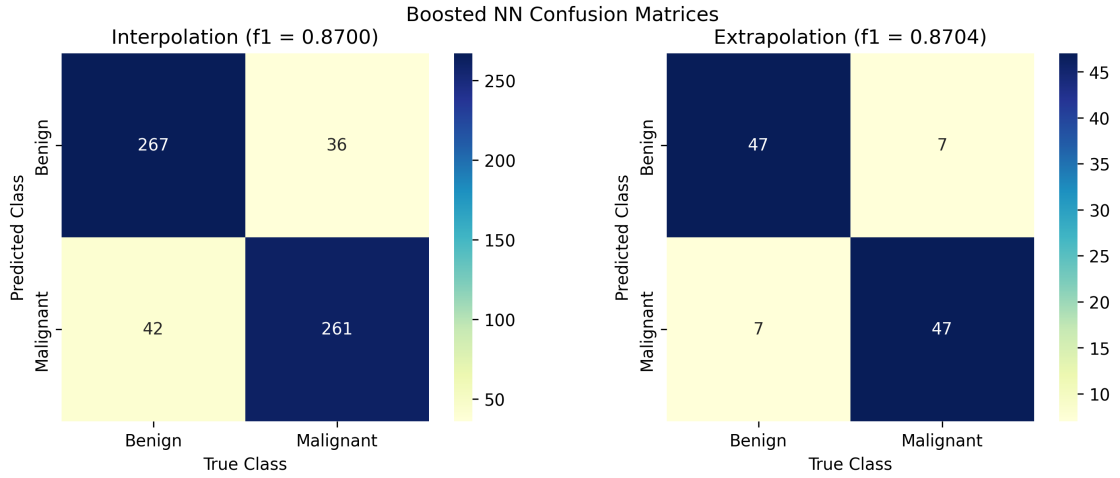
14 Model 10: Adaptive-Boosted Neural Network

[ ]:
```
class BaseClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, numClasses, optimizer, activation = 'relu'):
        self.numClasses = numClasses
        self.activation = activation
        self.optimizer = optimizer
        self.model = None
        self.classes_ = None

    def fit(self, X, y, sample_weight = None):
        self.classes_ = np.unique(y)
        numClasses = self.numClasses

        self.model = Sequential()
        self.model.add(Dense(100, activation = self.activation))
        self.model.add(Dense(75, activation = self.activation))
        self.model.add(Dense(50, activation = self.activation))
        self.model.add(Dense(25, activation = self.activation))
        self.model.add(Dense(20, activation = self.activation))
        self.model.add(Dense(numClasses, activation = 'softmax'))

        self.model.compile(loss = 'categorical_crossentropy', optimizer = self.
 ↪optimizer, metrics = ['accuracy'])

        y_one_hot = np.squeeze(np.eye(numClasses)[y])

        self.model.fit(X, y_one_hot, sample_weight = sample_weight, epochs =␣
 ↪20, verbose = 0)

        return self

    def predict_proba(self, X):
        return self.model.predict(X, verbose = 0)

    def predict(self, X):
        return np.argmax(self.predict_proba(X), axis = 1)
```

[ ]:
```
nnBoost = ABC (
    estimator = BaseClassifier (
        numClasses = 2,
        optimizer = Adam(learning_rate = 0.0005)
    )
)

nnBoost.fit(xTrain, yTrain)
```

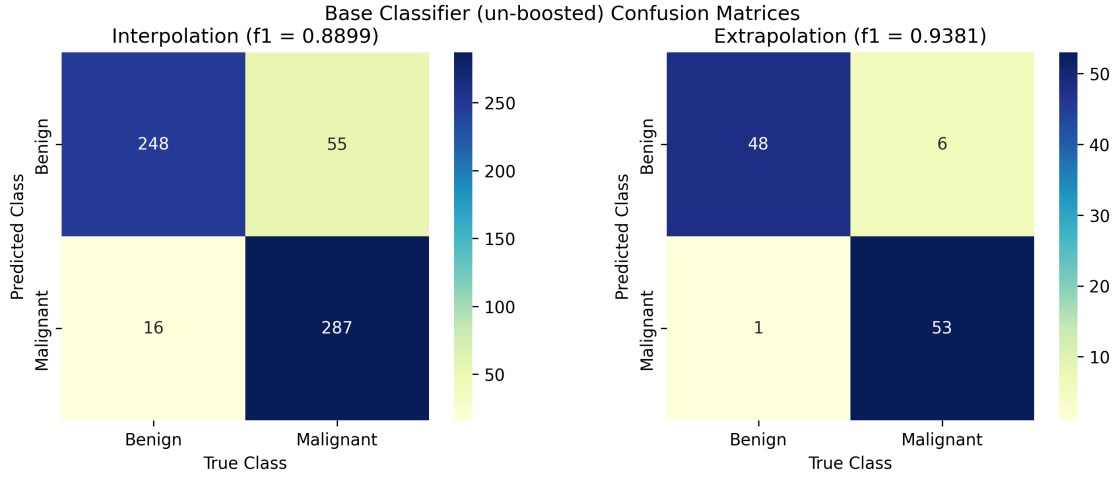
[ ]:
```
AdaBoostClassifier(estimator=BaseClassifier(numClasses=2,
optimizer=<keras.src.optimizers.legacy.adam.Adam object at 0x2c473fd60>))
```

[ ]:
```
evaluate(nnBoost, 'Boosted NN')
accuracy_score(yTest, nnBoost.predict(xTest))
```



[ ]:
```
0.8703703703703703
```

[ ]:
```
base = BaseClassifier(2, Adam(0.0005))
base.fit(xTrain, yTrain)
evaluate(base, 'Base Classifier (un-boosted)')
accuracy_score(yTest, base.predict(xTest))
```



[ ]:
```
0.9351851851851852
```

15 Image Dataset

[ ]:
```
# Extracting classes and training data from directory 'lung_image_sets'
train, test = image_dataset_from_directory (
    directory = '/Users/ericssonlin/code/Python/Vogelsberger/final/lc25000/
 ↪lung_colon_image_set/lung_image_sets',
    labels = 'inferred',
    label_mode = 'categorical',
    batch_size = 32,
    image_size = (384, 384),
    shuffle = True,
    seed = 42,
    validation_split = 0.125,
    subset = 'both'
)
```

```
Found 15000 files belonging to 3 classes.
Using 13125 files for training.
Using 1875 files for validation.
```

16 Model 1: CNN

[ ]:
```
# Classifier class
class CNNClassifier:
    """
    The `Classifier` class is made for single-class classification tasks
    (those that use categorical cross-entropy as an error function).

    # Methods
    - `__init__`: constructor
    - `loadData`: data loading tool
    - `loadSplittedData`: data loading tool
    - `preprocess`: loaded data preprocessing (RGB regularization to [0, 1] and␣
 ↪one-hot encoding)
    - `construct`: layer construction
    - `train`: model training
    - `evaluate`: model evaluation

    # Attributes
    - `inputShape`: shape of the input image
    - `outputClassCount`: number of output classes
    - `convolutionLayerCount`: number of convolution layers
    - `convolutionDropoutRates`: dropout layer parameters
    - `convolutionFilters`: channel expansion
    - `convolutionKernelSizes`: kernel size
    - `convolutionStrides`: downscaling
    - `poolSizes`: kernel size
```

```
    - `poolStrides`: downscaling
    - `denseLayerCount`: number of dense layers
    - `denseUnits`: number of neurons
    - `denseDropoutRates`: dropout layer parameters
    - `trainingSize`: training proportion
    - `testingSize`: testing proportion
    - `validationSize`: validation proportion
    - `layers`: actual layers (normalization, batch, etc., not included)
    - `input`: input layer
    - `output`: output layer
    - `xTrain, xTest, xValidate, yTrain, yTest, yValidate`: splitted data
    - `classifier`: actual classifier (keras Model object)
    - `srand`: random seed
    """
    def __init__ (
        self,
        inputShape: Tuple[int, int, int],
        outputClassCount: int,
        convolutionLayerCount: int,
        convolutionDropoutRates: List[float],
        convolutionFilters: List[int],
        convolutionKernelSizes: List[int],
        convolutionStrides: List[int],
        poolSizes: List[int],
        poolStrides: List[int],
        denseLayerCount: int,
        denseUnits: List[int],
        denseDropoutRates: List[float],
        trainingSize: float = 0.6,
        testingSize: float = 0.2,
        validationSize: float = 0.2,
        srand: int = 42
    ):
        """
        Constructor for classifier.
        # Parameters
        - `inputShape` is the input image shape
        - `outputClassCount` is the number of output classes being classified
        - `convolutionLayerCount` is the number of convolution layers in the␣
↪model
        - `convolutionDropoutRates` is the dropout rate of each convolution␣
↪layer
        - `convolutionFilters` is the number of filters (channel expansion) in␣
↪each convolution layer
        - `convolutionStrides` gives the downscale factor of each convolution␣
↪layer
        - `poolSizes` gives the sizes of the max-pooling windows
```

```
        - `poolStrides` gives the downscale factor of each pooling layer
        - `denseLayerCount` gives the number of dense layers
        - `denseUnits` gives the number of neurons in each dense layer
        - `denseDropoutRates` gives the dropout rate for each dense layer
        - `trainingSize` gives the proportion of data used for training
        - `testingSize` gives the proportion of data used for testing
        - `validationSize` gives the proportion of data used for validation
        - `srand` gives the randomization seed

        # Constraints
        - The product of the strides arrays must evenly divide the dimensions␣
↪of each image
        - Each list of layer parameters must match the corresponding layer␣
↪count filter
        - The sum of data proportions must equal 1 (all data used) and there␣
↪must be no negative numbers
        """

        self.inputShape = inputShape
        self.outputClassCount = outputClassCount
        self.convolutionLayerCount = convolutionLayerCount
        self.convolutionDropoutRates = convolutionDropoutRates
        self.convolutionFilters = convolutionFilters
        self.convolutionKernelSizes = convolutionKernelSizes
        self.convolutionStrides = convolutionStrides
        self.poolSizes = poolSizes
        self.poolStrides = poolStrides
        self.denseLayerCount = denseLayerCount
        self.denseUnits = denseUnits
        self.denseDropoutRates = denseDropoutRates
        self.trainingSize = trainingSize
        self.testingSize = testingSize
        self.validationSize = validationSize
        self.srand = srand
        self.layers = []

        assert(trainingSize + testingSize + validationSize == 1)
        assert(abs(trainingSize) + abs(testingSize) + abs(validationSize) == 1)
        assert(self.convolutionLayerCount > 0)
        assert(self.denseLayerCount > 0)
        assert(len(self.convolutionDropoutRates) == self.convolutionLayerCount)
        assert(len(self.convolutionFilters) == self.convolutionLayerCount)
        assert(len(self.convolutionKernelSizes) == self.convolutionLayerCount)
        assert(len(self.convolutionStrides) == self.convolutionLayerCount)
        assert(len(self.poolSizes) == self.convolutionLayerCount)
        assert(len(self.poolStrides) == self.convolutionLayerCount)
        assert(len(self.denseUnits) == self.denseLayerCount)
```

```
        assert(len(self.denseDropoutRates) == self.denseLayerCount)
        assert (
            self.inputShape[0] % (
                np.prod(self.convolutionStrides) * np.prod(self.poolStrides)
            ) == 0
        )
        assert (
            self.inputShape[1] % (
                np.prod(self.convolutionStrides) * np.prod(self.poolStrides)
            ) == 0
        )

        self._construct()

    def loadData(self, x, y, requiresResize: bool = True) -> None:
        """
        Load x and y data into the model, splitting into training,
        testing, and validation datasets as specified by the `trainingSize`,
        `testingSize`, and `validationSize` parameters set in the constructor
        """
        self.xTrain, xRemaining, self.yTrain, yRemaining = train_test_split (
            x,
            y,
            train_size = self.trainingSize,
            shuffle = True,
            stratify = y,
            random_state = self.srand
        )
        self.xTest, self.xValidate, self.yTest, self.yValidate =␣
↪train_test_split (
            xRemaining,
            yRemaining,
            train_size = self.testingSize / (1 - self.trainingSize),
            shuffle = True,
            stratify = yRemaining,
            random_state = self.srand
        )
        self._preprocess(requiresResize)
        return None

    def loadSplittedData(self, xTrain, yTrain, xTest, yTest, xValidate,␣
↪yValidate) -> None:
        """
        An alternate way to load already-split data into the model
        """
        self.xTrain, self.yTrain, self.xTest, self.yTest, self.xValidate, self.
↪yValidate = xTrain, yTrain, xTest, yTest, xValidate, yValidate
```

```
        self._preprocess()
        return None

    def _preprocess(self, requiresResize: bool = True) -> None:
        """
        Preprocesses all split data (x, y) so that x is a numpy array
        with all pixel values between 0 and 1 (normalized) and y is a
        numpy array in one-hot-encoded vector format (as expected for
        categorical data).\n
        This function is automatically called by the loading functions.
        """
        self.xTrain = np.array(self.xTrain)
        self.xTest = np.array(self.xTest)
        self.xValidate = np.array(self.xValidate)
        if requiresResize:
            for image in self.xTrain:
                image = tf.image.resize(image, (self.inputShape[0], self.
↪inputShape[1])).numpy()
            for image in self.xTest:
                image = tf.image.resize(image, (self.inputShape[0], self.
↪inputShape[1])).numpy()
            for image in self.xValidate:
                image = tf.image.resize(image, (self.inputShape[0], self.
↪inputShape[1])).numpy()
        self.yTrain = np.array(to_categorical(self.yTrain, self.
↪outputClassCount))
        self.yTest = np.array(to_categorical(self.yTest, self.outputClassCount))
        self.yValidate = np.array(to_categorical(self.yValidate, self.
↪outputClassCount))

    def _construct(self) -> None:
        """
        Construct layers as instructed by constructor parameters.\n
        Stores the layers in the `self.layers` list, as well as the input
        layer in `self.input` and output layer in `self.output`.\n
        Initializes a model stored in `self.classifier`.
        """
        self.layers.append (
            Input (shape = self.inputShape, name = "Input")
        )
        for convolutionLayer in range(self.convolutionLayerCount):
            self.layers.append (
                Dropout (
                    rate = self.convolutionDropoutRates[convolutionLayer],
                    name = f"Dropout_{convolutionLayer + 1}"
                ) (
                    LeakyReLU (name = f"LeakyReLU_{convolutionLayer + 1}") (
```

```
                        BatchNormalization (name =␣
↪f"Batch_Normalization_{convolutionLayer + 1}") (
                            MaxPool2D (
                                pool_size = self.poolSizes[convolutionLayer],
                                strides = self.poolStrides[convolutionLayer],
                                padding = 'same',
                                name = f"Maximum_Pool_{convolutionLayer + 1}"
                            ) (
                                Conv2D (
                                    filters = self.
↪convolutionFilters[convolutionLayer],
                                    kernel_size = self.
↪convolutionKernelSizes[convolutionLayer],
                                    strides = self.
↪convolutionStrides[convolutionLayer],
                                    padding = 'same',
                                    name = f"Convolution_{convolutionLayer + 1}"
                                ) (
                                    self.layers[-1]
                                )
                            )
                        )
                    )
                )
            )
        self.layers.append (
            Flatten (name = "Flatten") (
                self.layers[-1]
            )
        )
        for denseLayer in range (self.denseLayerCount):
            self.layers.append (
                Dropout (
                    rate = self.denseDropoutRates[denseLayer],
                    name = f"Dropout_{self.convolutionLayerCount + denseLayer +␣
↪1}"
                ) (
                    BatchNormalization (name = f"Batch_Normalization_{self.
↪convolutionLayerCount + denseLayer + 1}") (
                        LeakyReLU (name = f"Leaky_ReLU_{self.
↪convolutionLayerCount + denseLayer + 1}") (
                            Dense (
                                units = self.denseUnits[denseLayer],
                                name = f"Dense_{denseLayer + 1}"
                            ) (
                                self.layers[-1]
```

```
                        )
                    )
                )
            )
        )
        self.layers.append (
            Softmax (name = f"Softmax") (
                self.layers[-1]
            )
        )
        self.input = self.layers[0]
        self.output = self.layers[-1]
        self.classifier = Model (
            inputs = [self.input],
            outputs = [self.output]
        )
        return None

    def train (
        self,
        optimizer,
        epochs,
        callbacks,
        loadFile = None
    ):
        """
        Train `self.classifier` with loaded data.\n
        Expecting an optimizer (such as keras.optimizers.Adam), the number of␣
↪epochs
        to be passed, callbacks (to be passed to the keras fit function), and
        optionally a `.h5` file from which to load pre-trained model weights.
        """
        if loadFile != None:
            self.classifier.load_weights(loadFile)
        self.classifier.compile (
            optimizer = optimizer,
            loss = 'categorical_crossentropy',
            metrics = [
                'accuracy'
            ]
        )
        self.classifier.fit (
            x = self.xTrain,
            y = self.yTrain,
            shuffle = True,
            validation_data = (self.xValidate, self.yValidate),
            epochs = epochs,
```

```
            callbacks = callbacks
        )

    def evaluate(self) -> None:
        """
        Evaluate the trained classifier.
        """
        self.classifier.evaluate (
            self.xTest,
            self.yTest
        )
        return None

    pass

# Classifier instance
cnnClassifier = CNNClassifier (
    inputShape = (384, 384, 3), # 384x384 images, 3 color channels
    outputClassCount = 3, # lung_aca, lung_n, lung_scc
    convolutionLayerCount = 6,
    convolutionDropoutRates = [0.25, 0.45, 0.2, 0.35, 0.15, 0.2],
    convolutionFilters = [16, 32, 32, 64, 16, 32],
    convolutionKernelSizes = [1, 5, 2, 7, 3, 5],
    convolutionStrides = [1, 2, 2, 6, 1, 2],
    poolSizes = [2 for i in range(6)],
    poolStrides = [1 for i in range(6)],
    denseLayerCount = 2,
    denseUnits = [512, 3],
    denseDropoutRates = [0.1 for i in range(2)],
    trainingSize = 0.75,
    testingSize = 0.125,
    validationSize = 0.125,
    srand = 42
)
```

[ ]:
```
# The below code was used to train the classifier.
# It has been replaced by code loading a pretrained model saved by the below␣
 ↪code.
"""
cnnModel.compile (
    optimizer = Adam(learning_rate = 0.0005),
    metrics = ['accuracy'],
    loss = 'categorical_crossentropy'
)

cnnModel.fit (
    train,
```

```
    epochs = 200,
    callbacks = [
        ModelCheckpoint (
            "/Users/ericssonlin/code/Python/Vogelsberger/final/model4.keras",
            monitor = 'val_loss',
            save_best_only = True,
            save_weights_only = False,
            mode = 'min',
            verbose = 1
        ),
        EarlyStopping (
            monitor = 'val_loss',
            min_delta = 0.0001,
            patience = 5
        )
    ],
    validation_data = test
)

cnnModel.summary()
plot_model(cnnModel, 'model.png', show_shapes = True, dpi = 300)
"""

cnnClassifier.classifier = load_model('model4.keras')

cnnModel = cnnClassifier.classifier

cnnModel.summary()
plot_model(cnnModel, 'model.png', show_shapes = True, dpi = 300)
```

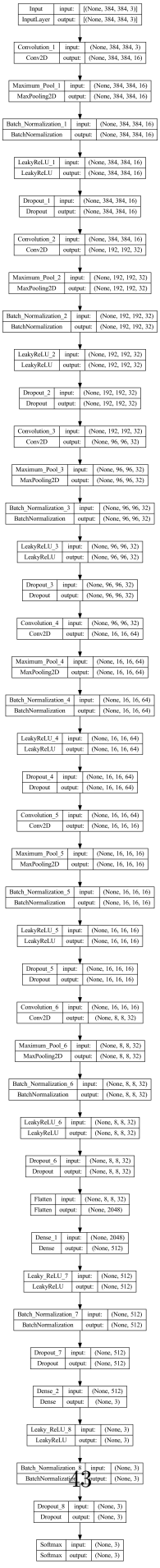
```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 Input (InputLayer)          [(None, 384, 384, 3)]     0

 Convolution_1 (Conv2D)      (None, 384, 384, 16)      64

 Maximum_Pool_1 (MaxPooling  (None, 384, 384, 16)      0
 2D)

 Batch_Normalization_1 (Bat  (None, 384, 384, 16)      64
 chNormalization)
```

```
LeakyReLU_1 (LeakyReLU)     (None, 384, 384, 16)      0

Dropout_1 (Dropout)         (None, 384, 384, 16)      0

Convolution_2 (Conv2D)      (None, 192, 192, 32)      12832

Maximum_Pool_2 (MaxPooling  (None, 192, 192, 32)      0
2D)

Batch_Normalization_2 (Bat  (None, 192, 192, 32)      128
chNormalization)

LeakyReLU_2 (LeakyReLU)     (None, 192, 192, 32)      0

Dropout_2 (Dropout)         (None, 192, 192, 32)      0

Convolution_3 (Conv2D)      (None, 96, 96, 32)        4128

Maximum_Pool_3 (MaxPooling  (None, 96, 96, 32)        0
2D)

Batch_Normalization_3 (Bat  (None, 96, 96, 32)        128
chNormalization)

LeakyReLU_3 (LeakyReLU)     (None, 96, 96, 32)        0

Dropout_3 (Dropout)         (None, 96, 96, 32)        0

Convolution_4 (Conv2D)      (None, 16, 16, 64)        100416

Maximum_Pool_4 (MaxPooling  (None, 16, 16, 64)        0
2D)

Batch_Normalization_4 (Bat  (None, 16, 16, 64)        256
chNormalization)

LeakyReLU_4 (LeakyReLU)     (None, 16, 16, 64)        0

Dropout_4 (Dropout)         (None, 16, 16, 64)        0

Convolution_5 (Conv2D)      (None, 16, 16, 16)        9232

Maximum_Pool_5 (MaxPooling  (None, 16, 16, 16)        0
2D)

Batch_Normalization_5 (Bat  (None, 16, 16, 16)        64
chNormalization)
```

```
 LeakyReLU_5 (LeakyReLU)     (None, 16, 16, 16)        0

 Dropout_5 (Dropout)         (None, 16, 16, 16)        0

 Convolution_6 (Conv2D)      (None, 8, 8, 32)          12832

 Maximum_Pool_6 (MaxPooling  (None, 8, 8, 32)          0
 2D)

 Batch_Normalization_6 (Bat  (None, 8, 8, 32)          128
 chNormalization)

 LeakyReLU_6 (LeakyReLU)     (None, 8, 8, 32)          0

 Dropout_6 (Dropout)         (None, 8, 8, 32)          0

 Flatten (Flatten)           (None, 2048)              0

 Dense_1 (Dense)             (None, 512)               1049088

 Leaky_ReLU_7 (LeakyReLU)    (None, 512)               0

 Batch_Normalization_7 (Bat  (None, 512)               2048
 chNormalization)

 Dropout_7 (Dropout)         (None, 512)               0

 Dense_2 (Dense)             (None, 3)                 1539

 Leaky_ReLU_8 (LeakyReLU)    (None, 3)                 0

 Batch_Normalization_8 (Bat  (None, 3)                 12
 chNormalization)

 Dropout_8 (Dropout)         (None, 3)                 0

 Softmax (Softmax)           (None, 3)                 0

=================================================================
Total params: 1192959 (4.55 MB)
Trainable params: 1191545 (4.55 MB)
Non-trainable params: 1414 (5.52 KB)
_________________________________________________________________
```

[ ]:


Input | input: | [(None, 384, 384, 3)]
InputLayer | output: | [(None, 384, 384, 3)]
Convolution_1 | input: | (None, 384, 384, 3)
Conv2D | output: | (None, 384, 384, 16)
Maximum_Pool_1 | input: | (None, 384, 384, 16)
MaxPooling2D | output: | (None, 384, 384, 16)
Batch_Normalization_1 | input: | (None, 384, 384, 16)
BatchNormalization | output: | (None, 384, 384, 16)
LeakyReLU_1 | input: | (None, 384, 384, 16)
LeakyReLU | output: | (None, 384, 384, 16)
Dropout_1 | input: | (None, 384, 384, 16)
Dropout | output: | (None, 384, 384, 16)
Convolution_2 | input: | (None, 384, 384, 16)
Conv2D | output: | (None, 192, 192, 32)
Maximum_Pool_2 | input: | (None, 192, 192, 32)
MaxPooling2D | output: | (None, 192, 192, 32)
Batch_Normalization_2 | input: | (None, 192, 192, 32)
BatchNormalization | output: | (None, 192, 192, 32)
LeakyReLU_2 | input: | (None, 192, 192, 32)
LeakyReLU | output: | (None, 192, 192, 32)
Dropout_2 | input: | (None, 192, 192, 32)
Dropout | output: | (None, 192, 192, 32)
Convolution_3 | input: | (None, 192, 192, 32)
Conv2D | output: | (None, 96, 96, 32)
Maximum_Pool_3 | input: | (None, 96, 96, 32)
MaxPooling2D | output: | (None, 96, 96, 32)
Batch_Normalization_3 | input: | (None, 96, 96, 32)
BatchNormalization | output: | (None, 96, 96, 32)
LeakyReLU_3 | input: | (None, 96, 96, 32)
LeakyReLU | output: | (None, 96, 96, 32)
Dropout_3 | input: | (None, 96, 96, 32)
Dropout | output: | (None, 96, 96, 32)
Convolution_4 | input: | (None, 96, 96, 32)
Conv2D | output: | (None, 16, 16, 64)
Maximum_Pool_4 | input: | (None, 16, 16, 64)
MaxPooling2D | output: | (None, 16, 16, 64)
Batch_Normalization_4 | input: | (None, 16, 16, 64)
BatchNormalization | output: | (None, 16, 16, 64)
LeakyReLU_4 | input: | (None, 16, 16, 64)
LeakyReLU | output: | (None, 16, 16, 64)
Dropout_4 | input: | (None, 16, 16, 64)
Dropout | output: | (None, 16, 16, 64)
Convolution_5 | input: | (None, 16, 16, 64)
Conv2D | output: | (None, 16, 16, 16)
Maximum_Pool_5 | input: | (None, 16, 16, 16)
MaxPooling2D | output: | (None, 16, 16, 16)
Batch_Normalization_5 | input: | (None, 16, 16, 16)
BatchNormalization | output: | (None, 16, 16, 16)
LeakyReLU_5 | input: | (None, 16, 16, 16)
LeakyReLU | output: | (None, 16, 16, 16)
Dropout_5 | input: | (None, 16, 16, 16)
Dropout | output: | (None, 16, 16, 16)
Convolution_6 | input: | (None, 16, 16, 16)
Conv2D | output: | (None, 8, 8, 32)
Maximum_Pool_6 | input: | (None, 8, 8, 32)
MaxPooling2D | output: | (None, 8, 8, 32)
Batch_Normalization_6 | input: | (None, 8, 8, 32)
BatchNormalization | output: | (None, 8, 8, 32)
LeakyReLU_6 | input: | (None, 8, 8, 32)
LeakyReLU | output: | (None, 8, 8, 32)
Dropout_6 | input: | (None, 8, 8, 32)
Dropout | output: | (None, 8, 8, 32)
Flatten | input: | (None, 8, 8, 32)
Flatten | output: | (None, 2048)
Dense_1 | input: | (None, 2048)
Dense | output: | (None, 512)
Leaky_ReLU_7 | input: | (None, 512)
LeakyReLU | output: | (None, 512)
Batch_Normalization_7 | input: | (None, 512)
BatchNormalization | output: | (None, 512)
Dropout_7 | input: | (None, 512)
Dropout | output: | (None, 512)
Dense_2 | input: | (None, 512)
Dense | output: | (None, 3)
Leaky_ReLU_8 | input: | (None, 3)
LeakyReLU | output: | (None, 3)
Batch_Normalization_8 | input: | (None, 3)
BatchNormalization | output: | (None, 3)
Dropout_8 | input: | (None, 3)
Dropout | output: | (None, 3)
Softmax | input: | (None, 3)
Softmax | output: | (None, 3)

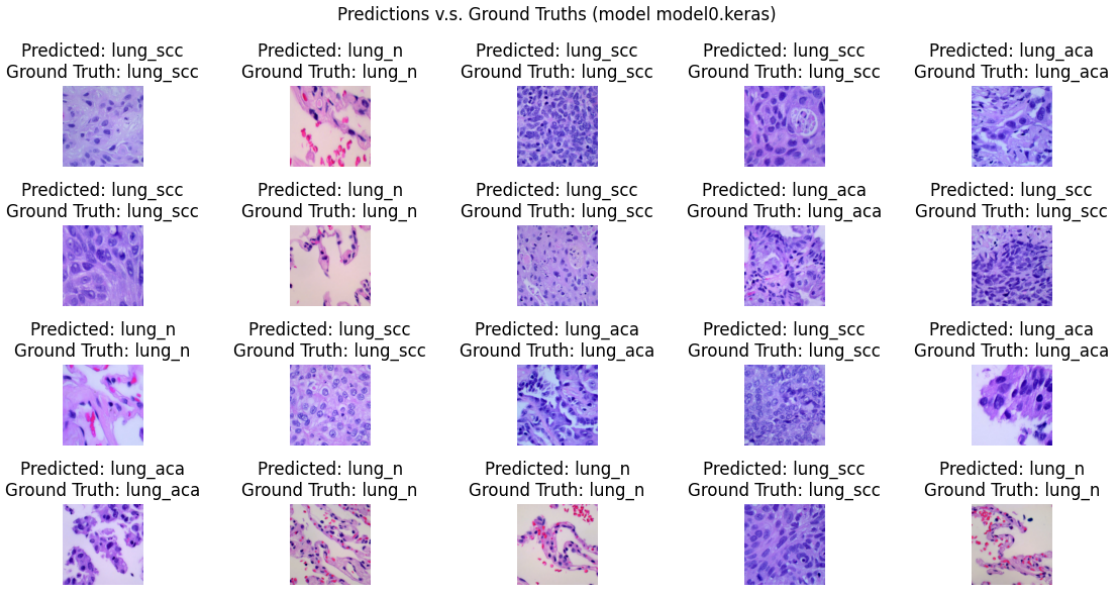
[ ]:
```
# Predict the test set
prediction = cnnModel.predict(test)

x = []
yTrue = []
progress = 0
names = test.class_names
for features, labels in test.take(20):
    print(f"Progress: {progress + 1}", end = '\r'); progress += 1
    for feature in features:
        x.append(feature.numpy())
    for label in labels:
        yTrue.append(label.numpy())
print("Converting images to numpy")
x = np.array(x)
imagesDisp = x / 255
print("Done converting images")
yTrue = np.array(yTrue)
yTrue = np.array([names[i] for i in np.argmax(yTrue, axis = 1)])

def evaluateTrainedModel(modelFile: str):
    pretrainedModel = load_model(modelFile)
    results = pretrainedModel.evaluate(test)
    yPred = pretrainedModel.predict(x)
    yPred = np.array([names[i] for i in np.argmax(yPred, axis = 1)])
    predictedTruthDisp = yPred
    groundTruthDisp = yTrue

    fig, axes = plt.subplots(nrows = 4, ncols = 5, figsize = (12, 6))
    fig.suptitle(f'Predictions v.s. Ground Truths (model {modelFile})')

    for i, ax in enumerate(axes.flatten()):
        ax.imshow(imagesDisp[i])
        ax.axis('off')
        ax.set_title(f'Predicted: {predictedTruthDisp[i]}\nGround Truth:␣
 ↪{groundTruthDisp[i]}')

    plt.tight_layout()
    plt.show()
    plt.clf()

    matrix = confusion_matrix(yTrue, yPred)
    sns.heatmap (
            matrix,
            annot = True,
```

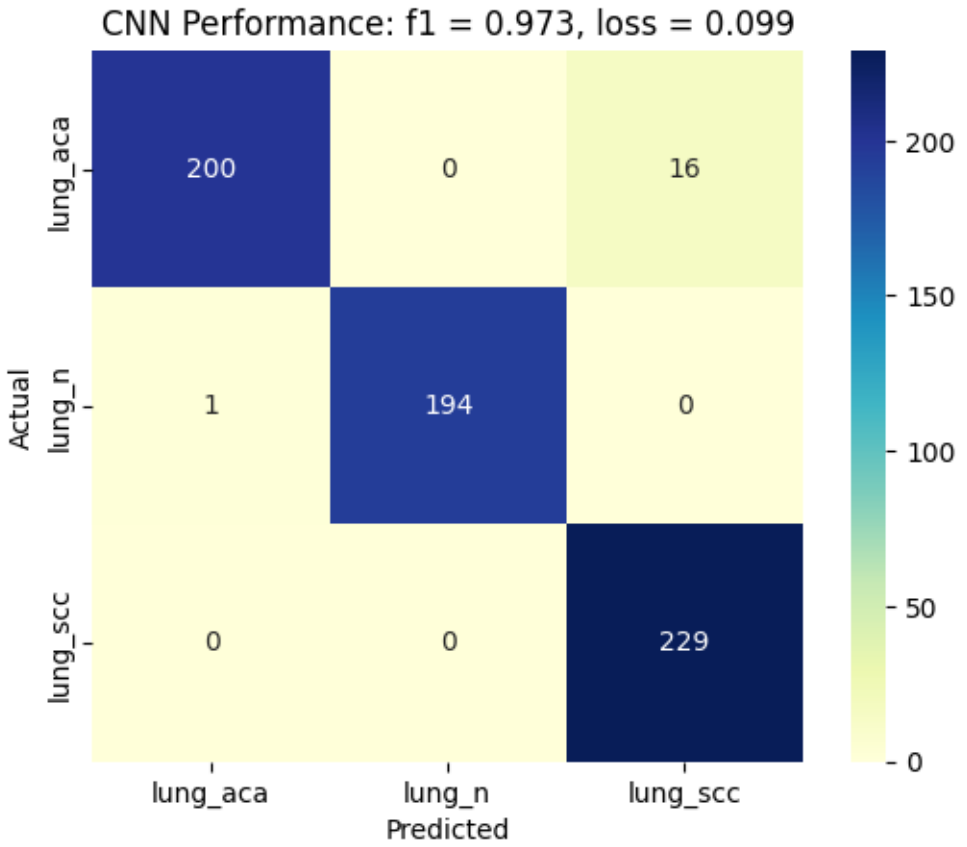
```
            fmt = 'd',
            cmap = 'YlGnBu',
            square = True,
            xticklabels = ['lung_aca', 'lung_n', 'lung_scc'],
            yticklabels = ['lung_aca', 'lung_n', 'lung_scc']
    )
    plt.title (
            'CNN Performance: f1 = %.3f, loss = %.3f' % (
                    f1_score(yTrue, yPred, average = "weighted"),
                    results[0]
            )
    )
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()
    plt.clf()
    return f1_score(yTrue, yPred, average = 'weighted'), results[0]
```

```
59/59 [==============================] - 14s 221ms/step
Converting images to numpy
Done converting images
```

[ ]:
```
f1, loss = [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]
f1[0], loss[0] = evaluateTrainedModel('model0.keras')
f1[1], loss[1] = evaluateTrainedModel('model1.keras')
f1[2], loss[2] = evaluateTrainedModel('model2.keras')
f1[3], loss[3] = evaluateTrainedModel('model3.keras')
f1[4], loss[4] = evaluateTrainedModel('model4.keras')
```

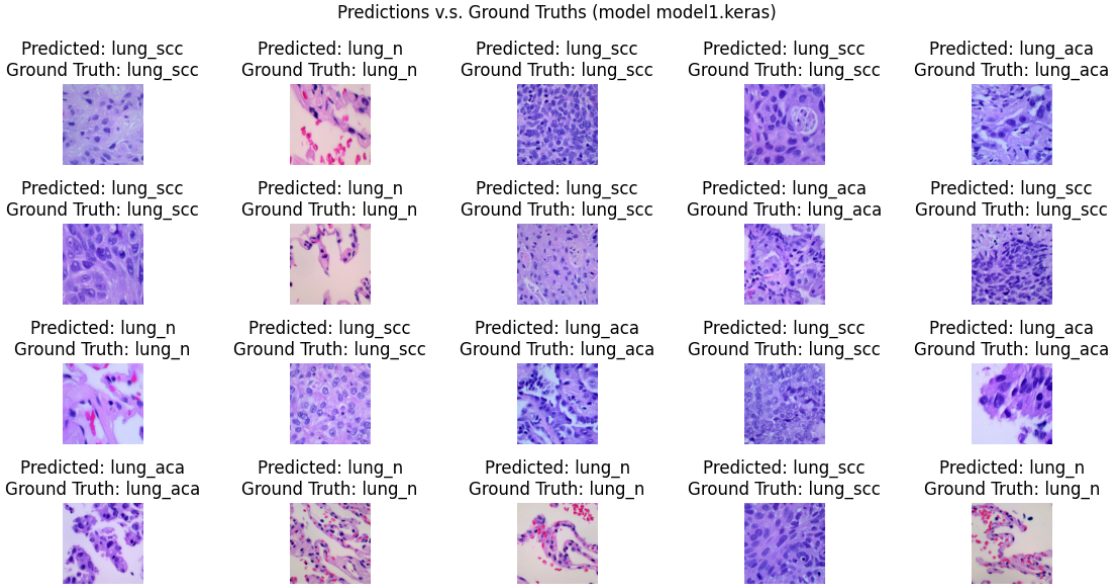
```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.

59/59 [==============================] - 13s 210ms/step - loss: 0.0990 -
accuracy: 0.9707
20/20 [==============================] - 3s 150ms/step
```

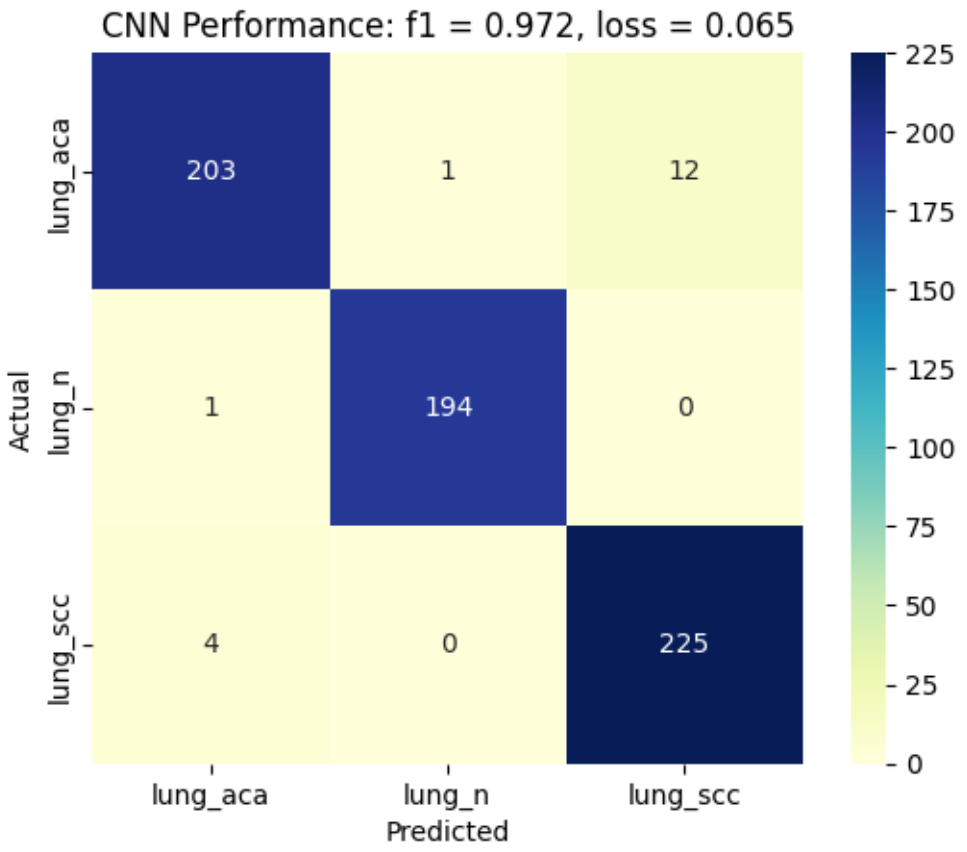




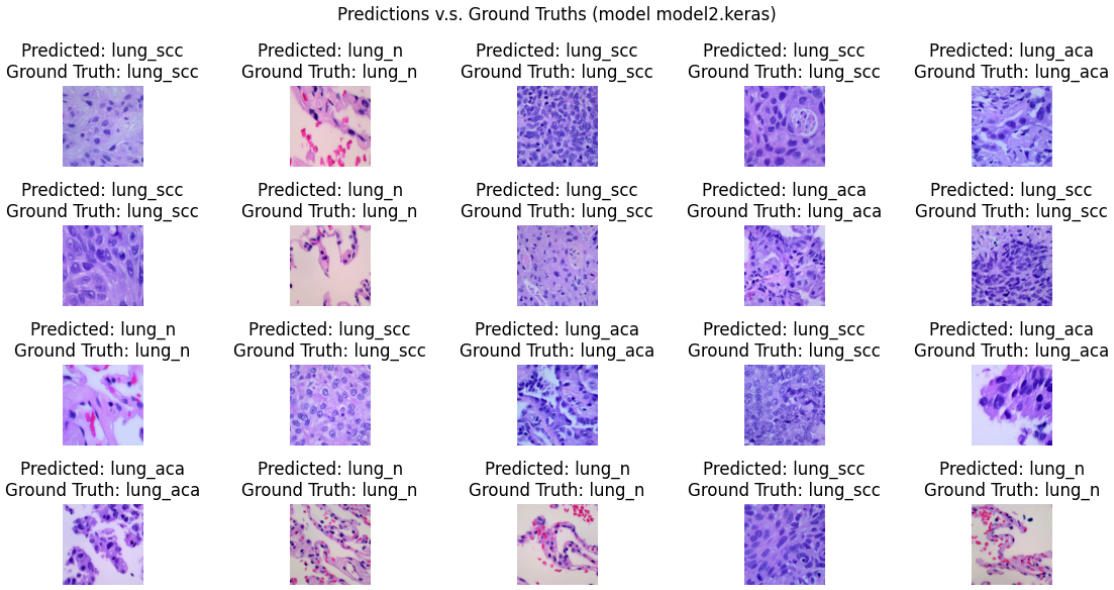
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs

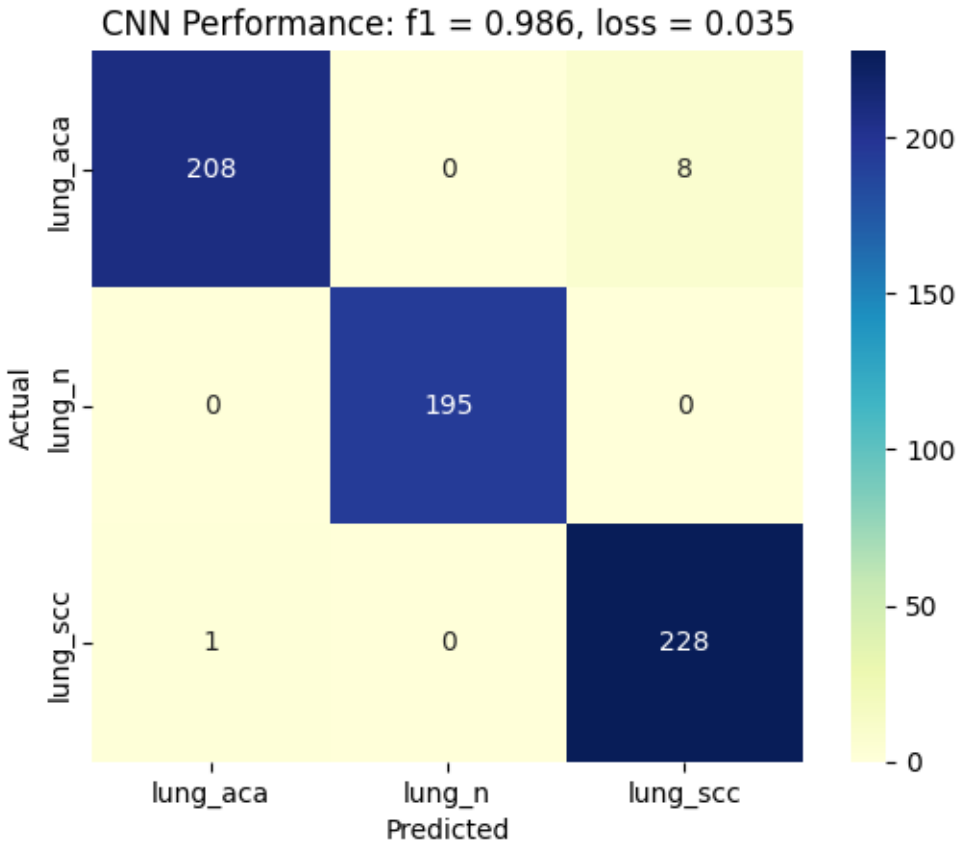
```
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.

59/59 [==============================] - 14s 232ms/step - loss: 0.0654 -
accuracy: 0.9771
20/20 [==============================] - 4s 175ms/step

<Figure size 640x480 with 0 Axes>
```





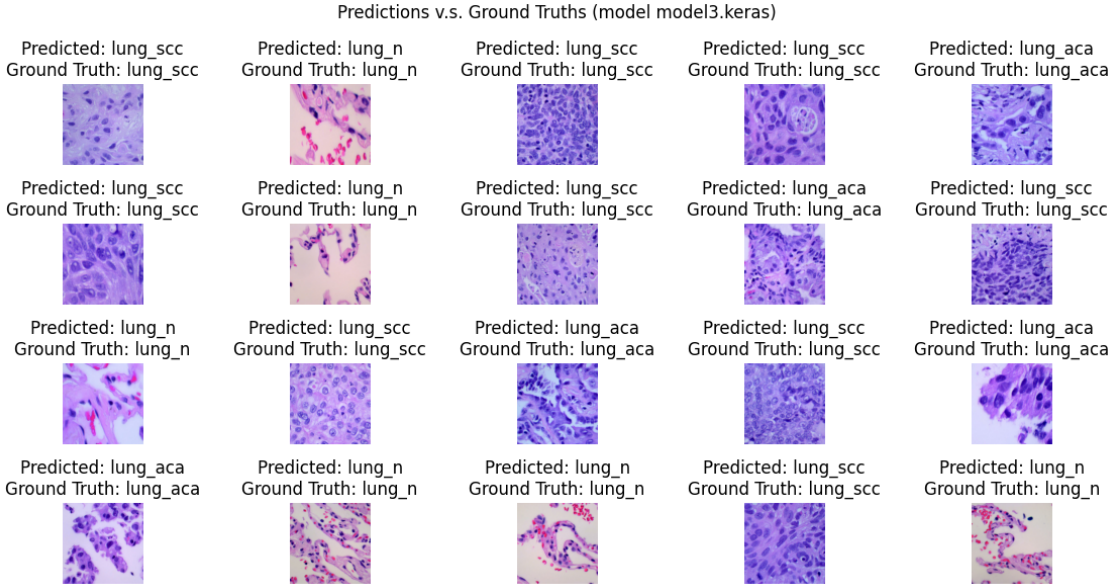

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.

59/59 [==============================] - 13s 215ms/step - loss: 0.0348 -
accuracy: 0.9872
20/20 [==============================] - 3s 147ms/step

<Figure size 640x480 with 0 Axes>
```

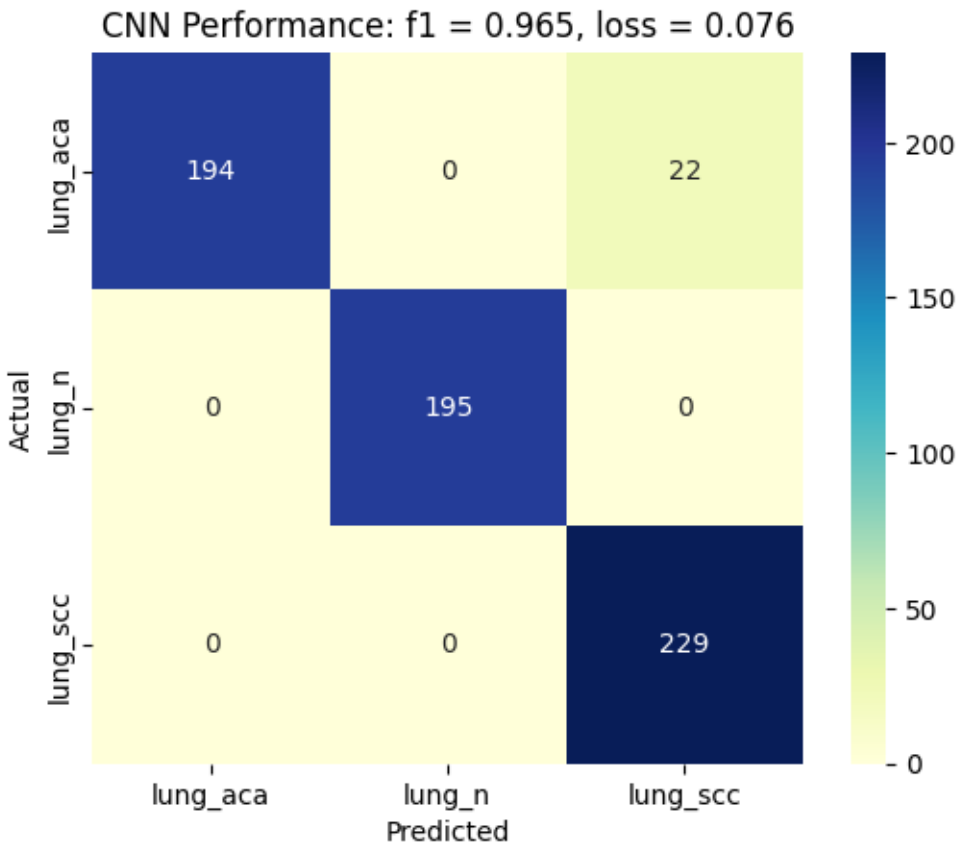




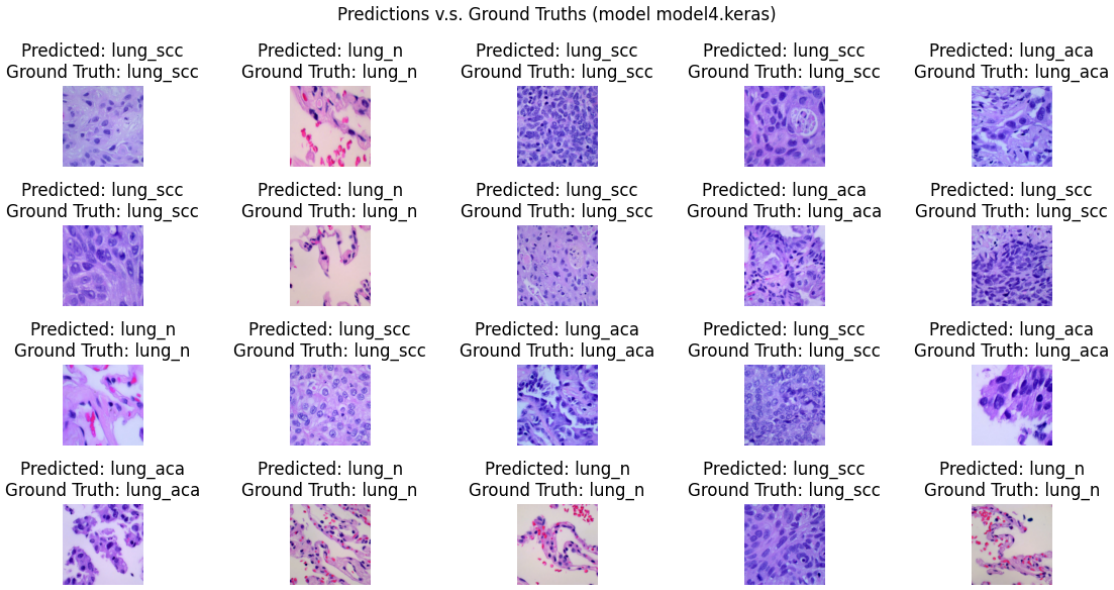

```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
```

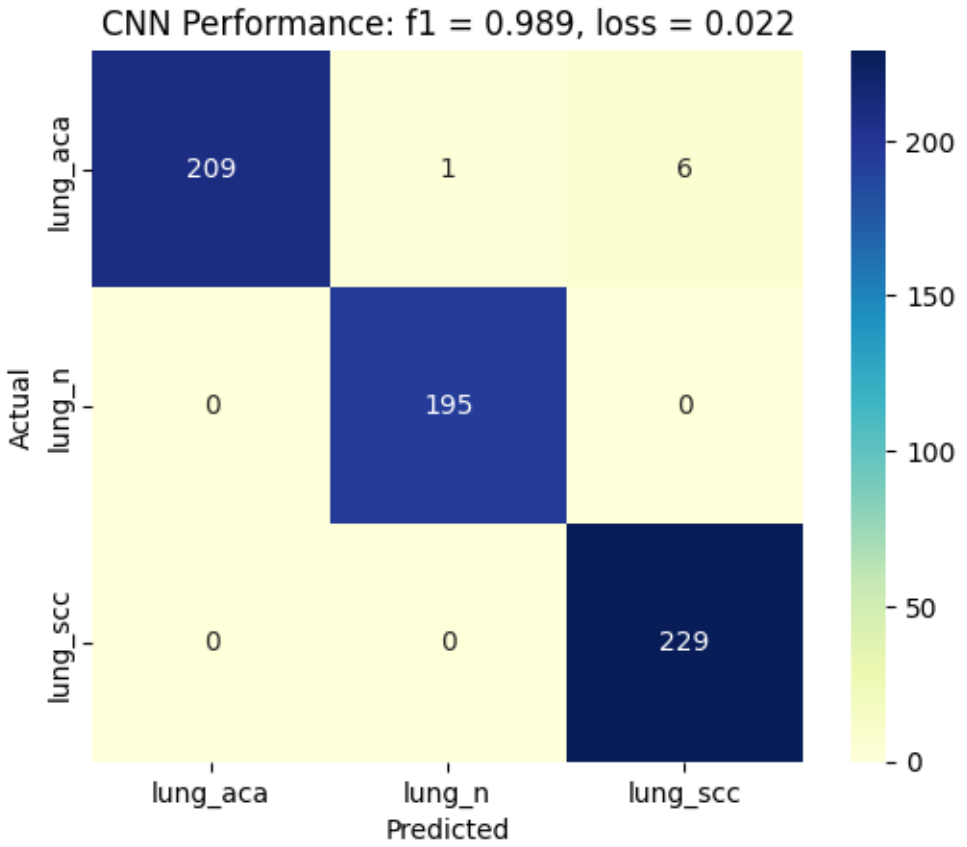
```
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.

59/59 [==============================] - 14s 221ms/step - loss: 0.0760 -
accuracy: 0.9691
20/20 [==============================] - 4s 169ms/step

<Figure size 640x480 with 0 Axes>
```






```
WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs
slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at
`tf.keras.optimizers.legacy.Adam`.

59/59 [==============================] - 14s 222ms/step - loss: 0.0223 -
accuracy: 0.9925
20/20 [==============================] - 5s 171ms/step

<Figure size 640x480 with 0 Axes>
```





<Figure size 640x480 with 0 Axes>

17 Model 2: Gradient Boosting on CNN-extracted Features

[ ]:
```
intermediate = Model(cnnClassifier.input, cnnClassifier.layers[8])

denseVal = intermediate.predict(train)
probaVal = cnnClassifier.classifier.predict(train)

denseVal[0], probaVal[0], len(denseVal[0]), len(probaVal[0])

dense = np.transpose(denseVal)
proba = np.transpose(probaVal)
```

```
411/411 [==============================] - 86s 208ms/step
411/411 [==============================] - 86s 209ms/step
```

[ ]:
```
yTrue = []
for feature, labels in train:
    for label in labels:
        yTrue.append(label.numpy())
yTrue = np.array([names[i] for i in np.argmax(np.array(yTrue), axis = 1)])

data = {
    'diagnosis': yTrue,
    **{i: dense[i] for i in range(512)},
    **{i + 512: proba[i] for i in range(3)}
}

cnnFeatures = pd.DataFrame(data = data)
```

[ ]:
```
cnnFeatures.info()
cnnFeatures.head()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13125 entries, 0 to 13124
Columns: 516 entries, diagnosis to 514
dtypes: float32(515), object(1)
memory usage: 25.9+ MB
```

[ ]:

	diagnosis	0	1	2	3	4	5	\
0	lung_aca	14.660162	1.470703	-4.265596	0.772520	6.315074	6.149839	
1	lung_scc	13.711671	-0.077413	-3.985240	3.464746	4.095089	6.980220	
2	lung_scc	14.002374	0.975688	-4.944846	1.869418	3.966516	7.029270	
3	lung_aca	13.590571	0.845271	-4.292871	0.740064	4.422274	4.331511	
4	lung_aca	12.352730	-0.675554	-4.357766	3.473469	6.334209	4.489475	

	6	7	8	…	505	506	507	\
0	5.502283	26.704390	-2.438900	…	-6.655910	-4.576254	-7.585865	

```
1  7.003735  26.175215 -3.028907  … -7.771262 -5.399694 -8.451774
2  7.722898  26.944851 -2.262516  … -6.727243 -4.734822 -7.952808
3  7.387825  25.303465 -3.108564  … -6.609543 -4.733815 -8.287434
4  7.916472  23.599281 -3.608661  … -6.924364 -4.964330 -6.966419

         508       509        510       511       512       513       514
0   9.073550 -5.586364  40.035603 -5.037138  0.000688  0.000026  0.999287
1   7.033656 -5.969585  45.091690 -4.690454  0.000230  0.000126  0.999644
2   9.345787 -4.796397  40.293865 -5.583335  0.999817  0.000020  0.000162
3   9.528415 -5.303717  39.784294 -3.876112  0.000085  0.000009  0.999906
4  10.608487 -5.026682  40.469574 -3.812705  0.995499  0.000431  0.004070

[5 rows x 516 columns]
```

[ ]:
```
xTrain, yTrain = cnnFeatures.iloc[:,1:], cnnFeatures.iloc[:,0]
```

[ ]:
```
gradientBoostingParams = {
    'loss': ['log_loss', 'exponential'],
    'n_estimators': np.arange(50, 200, 5),
    'learning_rate': np.linspace(0.4, 1.0, 20),
    'min_samples_leaf': np.linspace(0.01, 0.3, 50),
    'max_depth': [2, 3, 4, 5],
    'random_state': [42]
}

gradient = GBC(**{'random_state': 42, 'n_estimators': 135, 'min_samples_leaf':␣
 ↪0.2763265306122449, 'max_depth': 3, 'loss': 'log_loss', 'learning_rate': 0.
 ↪7789473684210526})

gradient.fit(xTrain, yTrain)
```

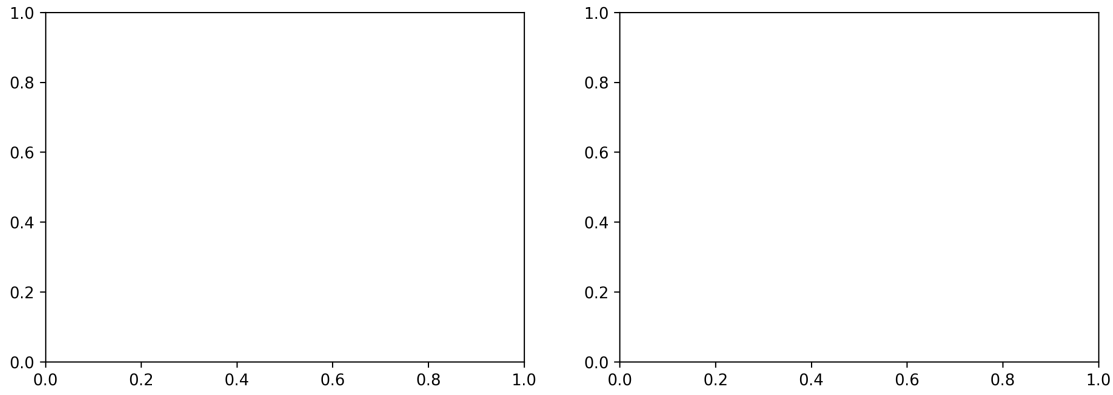
[ ]:
```
GradientBoostingClassifier(learning_rate=0.7789473684210526,
                           min_samples_leaf=0.2763265306122449,
                           n_estimators=135, random_state=42)
```

[ ]:
```
def getFeatures(x):
    yActual = []
    for feature, labels in x:
        for label in labels:
            yActual.append(label.numpy())
    yActual = np.array([names[i] for i in np.argmax(np.array(yActual), axis =␣
 ↪1)])
    densePred = np.transpose(intermediate.predict(x))
    probaPred = np.transpose(cnnClassifier.classifier.predict(x))
    dfData = {
        'diagnosis': yActual,
        **{i: densePred[i] for i in range(512)},
```

```
        **{i + 512: probaPred[i] for i in range(3)}
    }
    return pd.DataFrame(data = dfData)
```

[ ]:
```
trainingFeatures = getFeatures(train)
testingFeatures = getFeatures(test)
trainingPrediction = gradient.predict(trainingFeatures.iloc[:,1:])
testingPrediction = gradient.predict(testingFeatures.iloc[:,1:])
```

```
411/411 [==============================] - 95s 231ms/step
411/411 [==============================] - 89s 215ms/step
59/59 [==============================] - 12s 206ms/step
59/59 [==============================] - 13s 220ms/step
```



[ ]:
```
plt.clf()
fig, (ax1, ax2) = plt.subplots (
    1, 2,
    figsize = [12, 4],
    dpi = 300,
    clear = True
)
interpolationF1 = f1_score(trainingFeatures.iloc[:,0], trainingPrediction,␣
 ↪average = 'weighted')
extrapolationF1 = f1_score(testingFeatures.iloc[:,0], testingPrediction,␣
 ↪average = 'weighted')
fig.suptitle(f'Extracted-Feature Gradient Boosting Confusion Matrices')
ax1.title.set_text(('Interpolation (f1 = %.4f)' % (interpolationF1)))
ax2.title.set_text(('Extrapolation (f1 = %.4f)' % (extrapolationF1)))
interpolationConfusion = confusion_matrix(trainingFeatures.iloc[:,0],␣
 ↪trainingPrediction)
extrapolationConfusion = confusion_matrix(testingFeatures.iloc[:,0],␣
 ↪testingPrediction)
sns.heatmap (
```

```
    interpolationConfusion, annot = True, fmt = 'd',
    cmap = 'YlGnBu', ax = ax1, square = True,
    xticklabels = ['Benign', 'Malignant'],
    yticklabels = ['Benign', 'Malignant']
)
ax1.set(xlabel = "True Class", ylabel = "Predicted Class")
sns.heatmap (
    extrapolationConfusion, annot = True, fmt = 'd',
    cmap = 'YlGnBu', ax = ax2, square = True,
    xticklabels = ['Benign', 'Malignant'],
    yticklabels = ['Benign', 'Malignant']
)
ax2.set(xlabel = "True Class", ylabel = "Predicted Class")
plt.show()
```

```
<Figure size 640x480 with 0 Axes>
```