# 03f_LAB_Boosting_and_Stacking

May 18, 2022

## 1 Machine Learning Foundation

### 1.1 Course 3, Part f: Boosting and Stacking LAB

### 1.2 Introduction

We will be using the Human Activity Recognition with Smartphones database, which was built from the recordings of study participants performing activities of daily living (ADL) while carrying a smartphone with an embedded inertial sensors. The objective is to classify activities into one of the six activities (walking, walking upstairs, walking downstairs, sitting, standing, and laying) performed.

For each record in the dataset it is provided:

- Triaxial acceleration from the accelerometer (total acceleration) and the estimated body acceleration.
- Triaxial angular velocity from the gyroscope.
- A 561-feature vector with time and frequency domain variables.
- Its activity label.

More information about the features is available on the website shown above.

[2]:
```
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn

import seaborn as sns, pandas as pd, numpy as np
```

### 1.3 Question 1

- Import the data from the file `Human_Activity_Recognition_Using_Smartphones_Data.csv` and examine the shape and data types. For the data types, there will be too many to list each column separately. Rather, aggregate the types by count.
- Determine if the float columns need to be scaled.

[3]:
```
### BEGIN SOLUTION
data = pd.read_csv("https://cf-courses-data.s3.us.cloud-object-storage.
 ↪appdomain.cloud/IBM-ML241EN-SkillsNetwork/labs/datasets/
 ↪Human_Activity_Recognition_Using_Smartphones_Data.csv", sep=',')
```

The data has quite a few predictor columns.

[4]:
```
data.shape
```

[4]:
```
(10299, 562)
```

And they're all float values. The only non-float is the categories column, which is being predicted.

[4]:
```
data.dtypes.value_counts()
```

[4]:
```
float64    561
object       1
dtype: int64
```

The minimum and maximum value for the float columns is -1.0 and 1.0, respectively. However, scaling is never required for tree-based methods.

[5]:
```
# Mask to select float columns
float_columns = (data.dtypes == np.float)

# Verify that the maximum of all float columns is 1.0
print( (data.loc[:,float_columns].max()==1.0).all() )

# Verify that the minimum of all float columns is -1.0
print( (data.loc[:,float_columns].min()==-1.0).all() )
### END SOLUTION
```

```
True
True
```

### 1.4 Question 2

- Integer encode the activities.
- Split the data into train and test data sets. Decide if the data will be stratified or not during the train/test split.

[6]:
```
### BEGIN SOLUTION
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()

data['Activity'] = le.fit_transform(data['Activity'])

le.classes_
```

[6]:
```
array(['LAYING', 'SITTING', 'STANDING', 'WALKING', 'WALKING_DOWNSTAIRS',
       'WALKING_UPSTAIRS'], dtype=object)
```

[7]:
```
data.Activity.unique()
```

[7]:
```
array([2, 1, 0, 3, 4, 5])
```

**NOTE**: We are about to create training and test sets from `data`. On those datasets, we are going to run grid searches over many choices of parameters. This can take some time. In order to shorten the grid search time, feel free to downsample `data` and create `X_train, X_test, y_train, y_test` from the downsampled dataset.

Now split the data into train and test data sets. A stratified split was not used here. If there are issues with any of the error metrics on the test set, it can be a good idea to start model fitting over using a stratified split. Boosting is a pretty powerful model, though, so it may not be necessary in this case.

[8]:
```
from sklearn.model_selection import train_test_split

# Alternatively, we could stratify the categories in the split, as was done␣
 ↪previously
feature_columns = [x for x in data.columns if x != 'Activity']

X_train, X_test, y_train, y_test = train_test_split(data[feature_columns],␣
 ↪data['Activity'],
                test_size=0.3, random_state=42)
```

[9]:
```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
### END SOLUTION
```

[9]:
```
((7209, 561), (7209,), (3090, 561), (3090,))
```

### 1.5 Question 3

- Fit gradient boosted tree models with all parameters set to their defaults with the following tree numbers (`n_estimators = [15, 25, 50, 100, 200, 400]`) and evaluate the accuracy on the test data for each of these models.
- Plot the accuracy as a function of estimator number.

*Note:* there is no out-of-bag error for boosted models. And the `warm_flag=True` setting has a bug in the gradient boosted model, so don't use it. Simply create the model inside the `for` loop and set the number of estimators at this time. This will make the fitting take a little longer. Additionally, boosting models tend to take longer to fit than bagged ones because the decision stumps must be fit successively.

[10]:
```
### BEGIN SOLUTION
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

error_list = list()

# Iterate through various possibilities for number of trees
tree_list = [15, 25, 50, 100, 200, 400]
for n_trees in tree_list:
```

```
    # Initialize the gradient boost classifier
    GBC = GradientBoostingClassifier(n_estimators=n_trees, random_state=42)

    # Fit the model
    print(f'Fitting model with {n_trees} trees')
    GBC.fit(X_train.values, y_train.values)
    y_pred = GBC.predict(X_test)

    # Get the error
    error = 1.0 - accuracy_score(y_test, y_pred)

    # Store it
    error_list.append(pd.Series({'n_trees': n_trees, 'error': error}))

error_df = pd.concat(error_list, axis=1).T.set_index('n_trees')

error_df
```

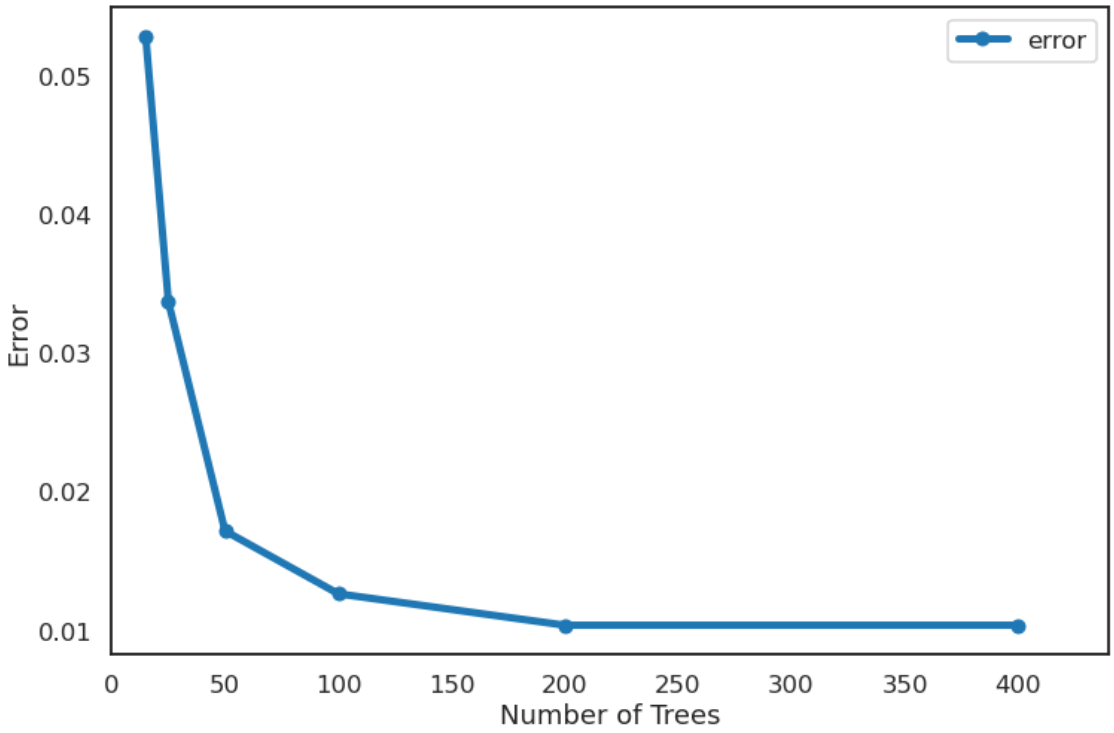
```
Fitting model with 15 trees
Fitting model with 25 trees
Fitting model with 50 trees
Fitting model with 100 trees
Fitting model with 200 trees
Fitting model with 400 trees
```

[10]:

| n_trees | error |
|---|---|
| 15.0 | 0.052751 |
| 25.0 | 0.033657 |
| 50.0 | 0.017152 |
| 100.0 | 0.012621 |
| 200.0 | 0.010356 |
| 400.0 | 0.010356 |

Now plot the result.

[11]:
```
sns.set_context('talk')
sns.set_style('white')

# Create the plot
ax = error_df.plot(marker='o', figsize=(12, 8), linewidth=5)

# Set parameters
ax.set(xlabel='Number of Trees', ylabel='Error')
ax.set_xlim(0, max(error_df.index)*1.1);
### END SOLUTION
```




### 1.6 Question 4

- Using a grid search with cross-validation, fit a new gradient boosted classifier with the same list of estimators as question 3. Also try varying the learning rates (0.1, 0.01, 0.001, etc.), the subsampling value (1.0 or 0.5), and the number of maximum features (1, 2, etc.).
- Examine the parameters of the best fit model.
- Calculate relevant error metrics on this model and examine the confusion matrix.

[12]:
```
### BEGIN SOLUTION
from sklearn.model_selection import GridSearchCV

# The parameters to be fit
param_grid = {'n_estimators': tree_list,
              'learning_rate': [0.1, 0.01, 0.001, 0.0001],
              'subsample': [1.0, 0.5],
              'max_features': [1, 2, 3, 4]}

# The grid search object
GV_GBC = GridSearchCV(GradientBoostingClassifier(random_state=42),
                      param_grid=param_grid,
                      scoring='accuracy',
                      n_jobs=-1)

# Do the grid search
```

```
GV_GBC = GV_GBC.fit(X_train, y_train)
```

[13]:
```
# The best model
GV_GBC.best_estimator_
```

[13]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=3,
              max_features=4, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=400,
              n_iter_no_change=None, presort='auto', random_state=42,
              subsample=0.5, tol=0.0001, validation_fraction=0.1,
              verbose=0, warm_start=False)
```

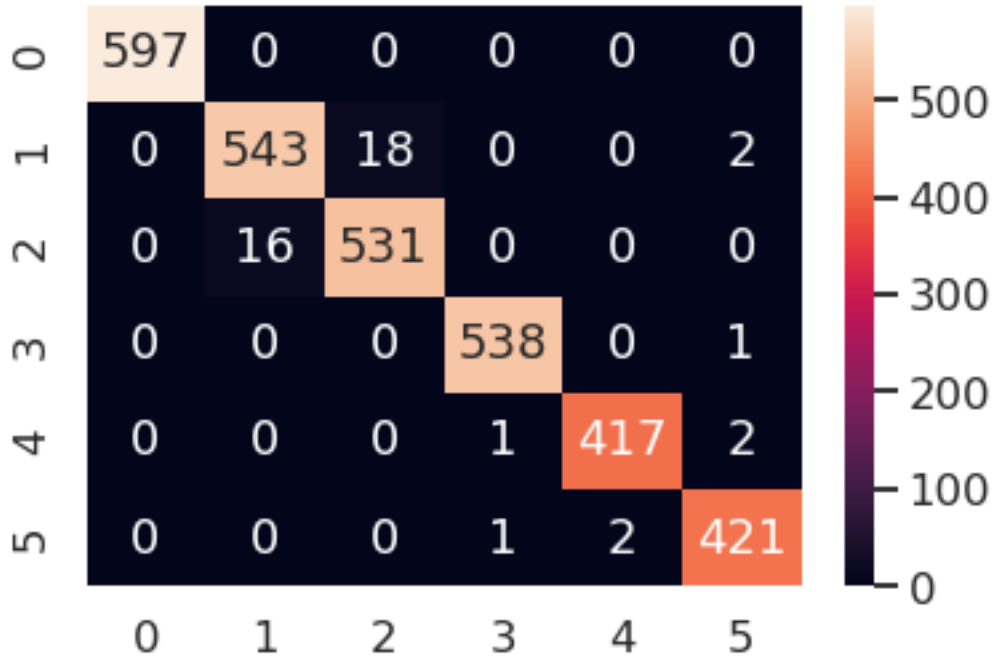
The error metrics. Classification report is particularly convenient for multi-class cases.

[14]:
```
from sklearn.metrics import classification_report

y_pred = GV_GBC.predict(X_test)
print(classification_report(y_pred, y_test))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       597
           1       0.96      0.97      0.97       559
           2       0.97      0.97      0.97       549
           3       1.00      1.00      1.00       540
           4       0.99      1.00      0.99       419
           5       0.99      0.99      0.99       426

   micro avg       0.99      0.99      0.99      3090
   macro avg       0.99      0.99      0.99      3090
weighted avg       0.99      0.99      0.99      3090
```

The confusion matrix. Note that the gradient boosted model has a little trouble distinguishing between activity class 1 and 2.

[15]:
```
from sklearn.metrics import confusion_matrix

sns.set_context('talk')
cm = confusion_matrix(y_test, y_pred)
ax = sns.heatmap(cm, annot=True, fmt='d')
### END SOLUTION
```




### 1.7 Question 5

- Create an AdaBoost model and fit it using grid search, much like question 4. Try a range of estimators between 100 and 200.
- Compare the errors from AdaBoost to those from the GradientBoostedClassifier.

[16]:
```
### BEGIN SOLUTION
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

ABC = AdaBoostClassifier(DecisionTreeClassifier(max_depth=1))

param_grid = {'n_estimators': [100, 150, 200],
              'learning_rate': [0.01, 0.001]}

GV_ABC = GridSearchCV(ABC,
                      param_grid=param_grid,
                      scoring='accuracy',
                      n_jobs=-1)

GV_ABC = GV_ABC.fit(X_train, y_train)
```

The best model.

[17]:
```
# The best model
GV_ABC.best_estimator_
```

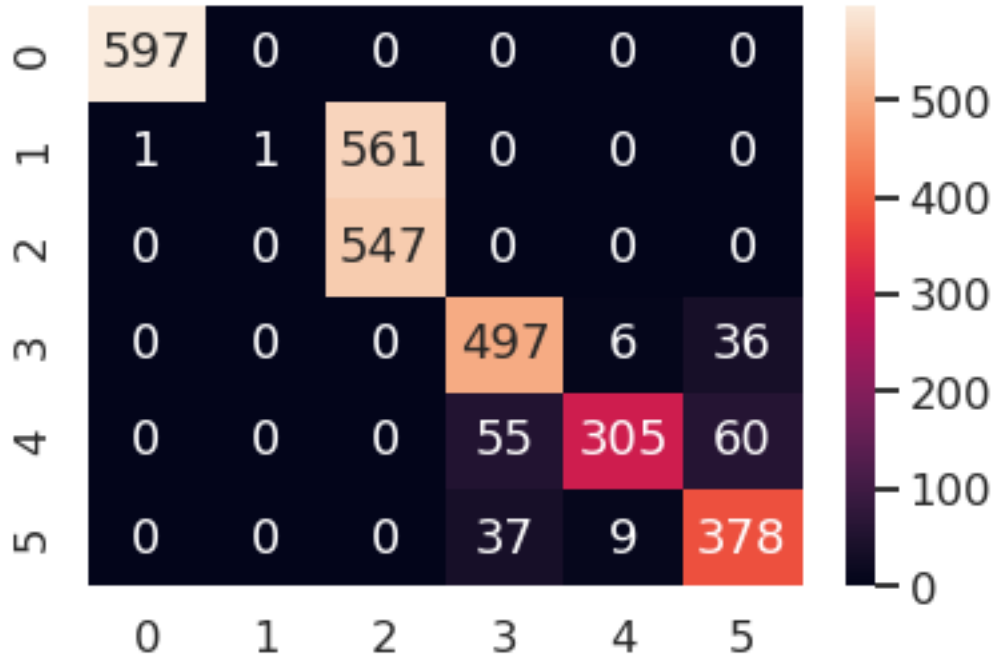
[17]:
```
AdaBoostClassifier(algorithm='SAMME.R',
          base_estimator=DecisionTreeClassifier(class_weight=None,
criterion='gini', max_depth=1,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best'),
          learning_rate=0.01, n_estimators=100, random_state=None)
```

The error metrics. Note that the issues with class 1 and 2 appear to have become more problematic. Also note other issues for classes 3 - 5. AdaBoost is very sensitive to outliers, so that could be the problem here.

[18]:
```
y_pred = GV_ABC.predict(X_test)
print(classification_report(y_pred, y_test))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       598
           1       0.00      1.00      0.00         1
           2       1.00      0.49      0.66      1108
           3       0.92      0.84      0.88       589
           4       0.73      0.95      0.82       320
           5       0.89      0.80      0.84       474

   micro avg       0.75      0.75      0.75      3090
   macro avg       0.76      0.85      0.70      3090
weighted avg       0.94      0.75      0.81      3090
```

[19]:
```
sns.set_context('talk')
cm = confusion_matrix(y_test, y_pred)
ax = sns.heatmap(cm, annot=True, fmt='d')
### END SOLUTION
```




### 1.8 Question 6

- Fit a logistic regression model with regularization.
- Using `VotingClassifier`, fit the logistic regression model along with either the GratientBoostedClassifier or the AdaBoost model (or both) from questions 4 and 5.
- Determine the error as before and compare the results to the appropriate gradient boosted model(s).
- Plot the confusion matrix for the best model created in this set of exercises.

[20]:
```
### BEGIN SOLUTION
from sklearn.linear_model import LogisticRegression

# L2 regularized logistic regression
LR_L2 = LogisticRegression(penalty='l2', max_iter=500, solver='saga').
 ↪fit(X_train, y_train)
```

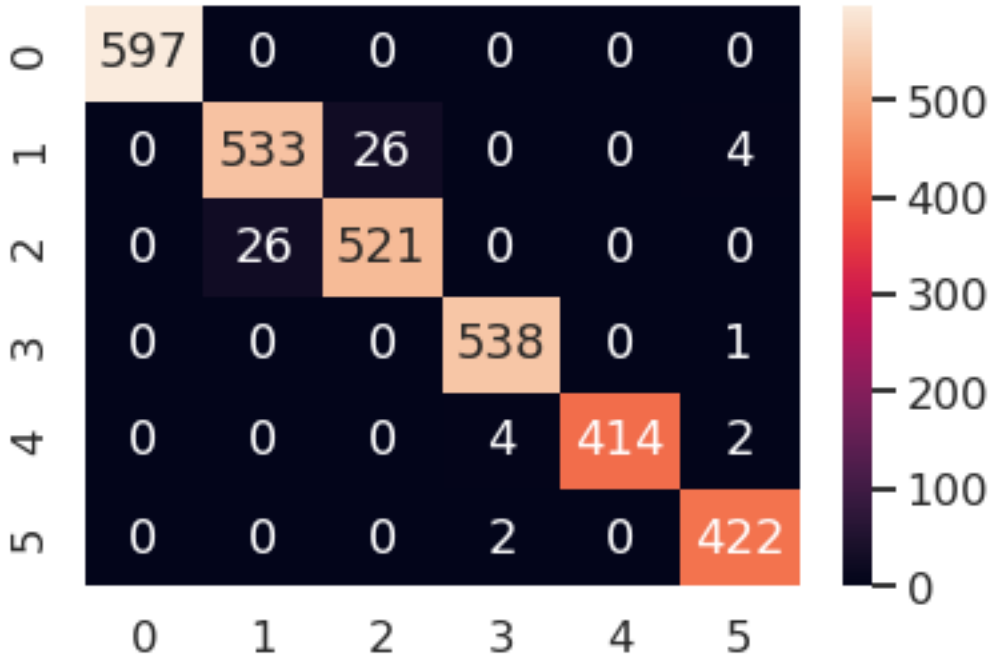
Check the errors and confusion matrix for the logistic regression model.

[21]:
```
y_pred = LR_L2.predict(X_test)
print(classification_report(y_pred, y_test))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       597
           1       0.95      0.95      0.95       559
           2       0.95      0.95      0.95       547
```

```
           3       1.00      0.99      0.99       544
           4       0.99      1.00      0.99       414
           5       1.00      0.98      0.99       429

   micro avg       0.98      0.98      0.98      3090
   macro avg       0.98      0.98      0.98      3090
weighted avg       0.98      0.98      0.98      3090
```

[22]:
```
sns.set_context('talk')
cm = confusion_matrix(y_test, y_pred)
ax = sns.heatmap(cm, annot=True, fmt='d')
```




And now the stacked model.

[23]:
```
from sklearn.ensemble import VotingClassifier

# The combined model--logistic regression and gradient boosted trees
estimators = [('LR_L2', LR_L2), ('GBC', GV_GBC)]

# Though it wasn't done here, it is often desirable to train
# this model using an additional hold-out data set and/or with cross validation
VC = VotingClassifier(estimators, voting='soft')
VC = VC.fit(X_train, y_train)
```

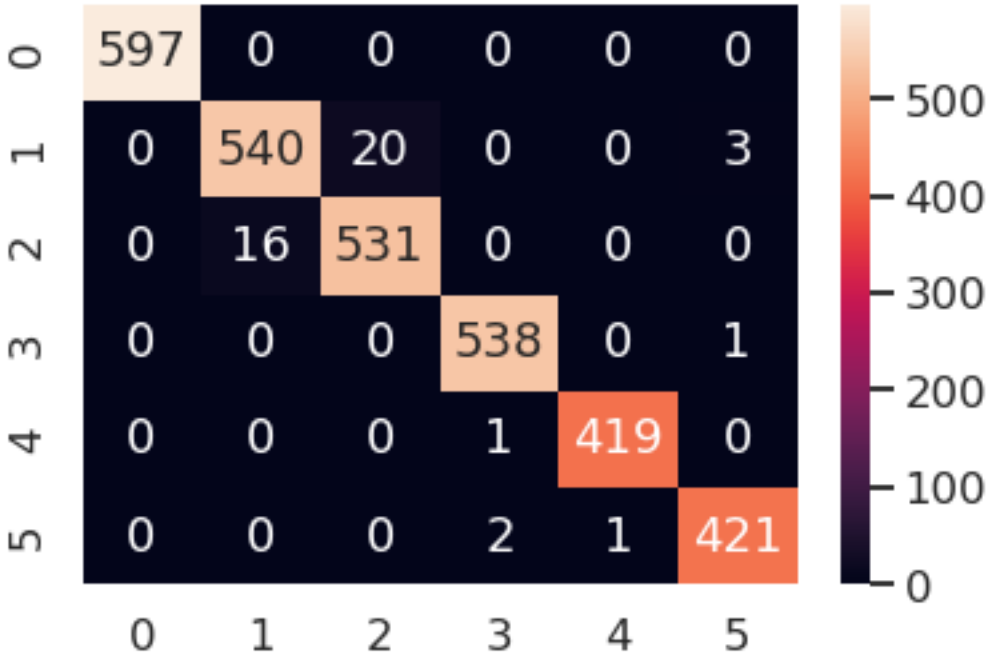
Performance for the voting classifier should improve relative to either logistic regression or gradi-

ent boosted trees alone. However, the fact that logistic regression does almost as well as gradient boosted trees is an important reminder to try the simplest model first. In some cases, its performance will be good enough.

[24]:
```
y_pred = VC.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       597
           1       0.97      0.96      0.97       563
           2       0.96      0.97      0.97       547
           3       0.99      1.00      1.00       539
           4       1.00      1.00      1.00       420
           5       0.99      0.99      0.99       424

   micro avg       0.99      0.99      0.99      3090
   macro avg       0.99      0.99      0.99      3090
weighted avg       0.99      0.99      0.99      3090
```

[25]:
```
sns.set_context('talk')
cm = confusion_matrix(y_test, y_pred)
ax = sns.heatmap(cm, annot=True, fmt='d')
### END SOLUTION
```



### 1.8.1 Machine Learning Foundation (C) 2020 IBM Corporation