Grading

The final score that you will receive for your programming assignment is generated in relation to the total points set in your programming assignment item—not the total point value in the nbgrader notebook. When calculating the final score shown to learners, the programming assignment takes the percentage of earned points vs. the total points provided by nbgrader and returns a score matching the equivalent percentage of the point value for the programming assignment.
DO NOT CHANGE VARIABLE OR METHOD SIGNATURES The autograder will not work properly if your change the variable or method signatures.

WARNING

Please refrain from using **print statements/anything that dumps large outputs(>500 lines) to STDOUT** to avoid running to into **memory issues**. Doing so requires your entire lab to be reset which may also result in loss of progress and you will be required to reach out to Coursera for assistance with this. This process usually takes time causing delays to your submission.

Validate Button

Please note that this assignment uses nbgrader to facilitate grading. You will see a **validate button** at the top of your Jupyter notebook. If you hit this button, it will run tests cases for the lab that aren't hidden. It is good to use the validate button before submitting the lab. Do know that the labs in the course contain hidden test cases. The validate button will not let you know whether these test cases pass. After submitting your lab, you can see more information about these hidden test cases in the Grader Output.
Cells with longer execution times will cause the validate button to time out and freeze. Please know that if you run into Validate time-outs, it will not affect the final submission grading.

Homework 5: Ensemble methods (adaBoost, random forests)

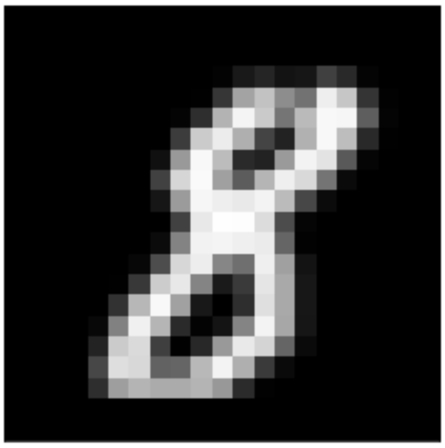
In [1]:
```
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.base import clone
from sklearn import tree
from sklearn.model_selection import train_test_split
import matplotlib.pylab as plt
%matplotlib inline
```

Run the helper code below to create a training and validation set of threes and eights from the MNIST dataset. There is also a helper function to display digits.

In [2]:
```
class ThreesandEights:
    """
    Class to store MNIST 3s and 8s data
    """

    def __init__(self, location):

        import pickle, gzip

        # Load the dataset
        f = gzip.open(location, 'rb')

        # Split the data set
        x_train, y_train, x_test, y_test = pickle.load(f)

        # Extract only 3's and 8's for training set
        self.x_train = x_train[np.logical_or(y_train== 3, y_train == 8),
:]
        self.y_train = y_train[np.logical_or(y_train== 3, y_train == 8)]
        self.y_train = np.array([1 if y == 8 else -1 for y in self.y_trai
n])

        # Shuffle the training data
        shuff = np.arange(self.x_train.shape[0])
        np.random.shuffle(shuff)
        self.x_train = self.x_train[shuff,:]
        self.y_train = self.y_train[shuff]

        # Extract only 3's and 8's for validation set
        self.x_test = x_test[np.logical_or(y_test== 3, y_test == 8), :]
        self.y_test = y_test[np.logical_or(y_test== 3, y_test == 8)]
        self.y_test = np.array([1 if y == 8 else -1 for y in self.y_tes
t])

        f.close()

def view_digit(ex, label=None, feature=None):
    """
    function to plot digit examples
    """
    if label: print("true label: {:d}".format(label))
    img = ex.reshape(21,21)
    col = np.dstack((img, img, img))
    if feature is not None: col[feature[0]//21, feature[0]%21, :] = [1, 0
, 0]
    plt.imshow(col)
    plt.xticks([]), plt.yticks([])

data = ThreesandEights("data/mnist21x21_3789.pklz")
```

```
In [3]: view_digit(data.x_train[0], data.y_train[0])
```

```
true label: 1
```



Problem 1: Building an Adaboost Classifier to classify MNIST digits 3 and 8.

Recall that the model we attempt to learn in AdaBoost is given by

$$H(\mathbf{x}) = \text{sign}\left[\sum_{k=1}^{K} \alpha_k h_k(\mathbf{x})\right]$$

where $h_k(\mathbf{x})$ is the k^{th} weak learner and α_k is it's associated ensemble coefficient.

Part A: [Peer Review, 5 pts] In the `AdaBoost` class below, implement the `fit` method to learn the sequence of weak learners $\{h_k(\mathbf{x})\}_{k=1}^K$ and corresponding coefficients $\{\alpha_k\}_{k=1}^K$. Note that you may use sklearn's implementation of DecisionTreeClassifier as your weak learner which allows you to pass as an optional parameter the weights associated with each training example. An example of instantiating and training a single learner is given in the comments of the `fit` method.

Recall that the AdaBoost algorithm is as follows:

```
 for k=1 to K:

     a) Fit kth weak learner to training data with weights w

     b) Computed weighted error errk for the kth weak learner (Check Adaboost slides
for formula).

     c) compute vote weight alpha[k] = 0.5 ln ((1-errk)/errk))

     d) update training example weights w[i] *= exp[-alpha[k] y[i] h[k](x[i])]

     e) normalize training weights so they sum to 1
```

For this week's Peer Review assignment, you wil upload a screenshot of the fit and error_rate functions from the AdaBoost class. Then you will upload a screenshot of using the fit function to fit the Adaboost classifier with 150 base decision stumps.

In [4]:
```
class AdaBoost:
    def __init__(self, n_learners=20, base=DecisionTreeClassifier(max_dep
th=3), random_state=1234):
        """
        Create a new adaboost classifier.

        Args:
            N (int, optional): Number of weak learners in classifier.
            base (BaseEstimator, optional): Your general weak learner
            random_state (int, optional): set random generator.  needed f
or unit testing.

        Attributes:
            base (estimator): Your general weak learner
            n_learners (int): Number of weak learners in classifier.
            alpha (ndarray): Coefficients on weak learners.
            learners (list): List of weak learner instances.
        """

        np.random.seed(42)

        self.n_learners = n_learners
        self.base = base
        self.alpha = np.zeros(self.n_learners)
        self.learners = []

    def fit(self, X_train, y_train):
        """
        Train AdaBoost classifier on data. Sets alphas and learners.

        Args:
            X_train (ndarray): [n_samples x n_features] ndarray of traini
ng data
            y_train (ndarray): [n_samples] ndarray of data
        """

        # ===============================================================
==
        # TODO

        # Note: You can create and train a new instantiation
        # of your sklearn decision tree as follows
        # you don't have to use sklearn's fit function,
        # but it is probably the easiest way

        # Note: Weights(w) should be initialized with dtype = np.float128
for higher precision
        # This is necessary for the sample test case to pass.

        # w = np.ones(len(y_train), dtype = np.float128)
        # w /= np.sum(w)
        # for loop
        #   h = clone(self.base)
        #   h.fit(X_train, y_train, sample_weight=w)
        #   ...
        #
        #
        #   ...
        #   Save alpha and learner
        # ===============================================================
==

        # your code here

        n_samples = X_train.shape[0]
        w = np.ones(n_samples, dtype=np.float128) / n_samples

        for k in range(self.n_learners):
            h = clone(self.base)
            h.fit(X_train, y_train, sample_weight=w)
            y_pred = h.predict(X_train)

            err = np.sum(w[y_pred != y_train])
            alpha_k = 0.5 * np.log((1 - err) / err)

            w *= np.exp(-alpha_k * y_train * y_pred)
            w /= np.sum(w)

            self.alpha[k] = alpha_k
            self.learners.append(h)

        return self

    def error_rate(self, y_true, y_pred, weights):
        """
        Compute the weighted error rate.

        Args:
            y_true (ndarray): [n_samples] ndarray of true labels
            y_pred (ndarray): [n_samples] ndarray of predicted labels
            weights (ndarray): [n_samples] ndarray of sample weights

        Returns:
            error (float): Weighted error rate
        """

        # Implement the weighted error rate
        return 1 - np.sum(weights * np.equal(y_true, y_pred)) / np.sum(we
ights)



    def predict(self, X):
        """
        Adaboost prediction for new data X.

        Args:
            X (ndarray): [n_samples x n_features] ndarray of data

        Returns:
            yhat (ndarray): [n_samples] ndarray of predicted labels
{-1,1}
        """

        # ===============================================================
==
        # TODO
        # ===============================================================
```

```python
==
        yhat = np.zeros(X.shape[0])

        # your code here
        for alpha, learner in zip(self.alpha, self.learners):
            yhat += alpha * learner.predict(X)

        return np.sign(yhat)


    def score(self, X, y):
        """
        Computes prediction accuracy of classifier.

        Args:
            X (ndarray): [n_samples x n_features] ndarray of data
            y (ndarray): [n_samples] ndarray of true labels

        Returns:
            Prediction accuracy (between 0.0 and 1.0).
        """

        # your code here
        y_pred = self.predict(X)
        return np.mean(y_pred == y)

    def staged_score(self, X, y):
        """
        Computes the ensemble score after each iteration of boosting
        for monitoring purposes, such as to determine the score on a
        test set after each boost.

        Args:
            X (ndarray): [n_samples x n_features] ndarray of data
            y (ndarray): [n_samples] ndarray of true labels

        Returns:
            scores (ndarary): [n_learners] ndarray of scores
        """

        scores = []


        y_pred = np.zeros(X.shape[0])

        for alpha, learner in zip(self.alpha, self.learners):
            y_pred += alpha * learner.predict(X)
            scores.append(np.mean(np.sign(y_pred) == y))

        return np.array(scores)
```

```
In [5]: # Sample test for Adaboost error rate function.
        import pytest

        y_true = [-1, 1, 1, -1, 1, -1, -1]
        y_pred = [-1, 1, 1, 1, 1, -1, 1]
        w = np.ones(len(y_true))
        w /= np.sum(w)

        clf = AdaBoost()
        err_rate = clf.error_rate(y_true, y_pred, w)
        assert pytest.approx(err_rate, 0.01) == 0.2857, "Check the error_rate fun
        ction."
```

```
In [6]: err_rate
```

```
Out[6]: 0.2857142857142857
```

```
In [7]: # Sample test for Adaboost fit function.

        sample_data = np.load('train.npz')
        sample_X = sample_data['X']
        sample_y = sample_data['y']
        test_model = AdaBoost(n_learners=5).fit(sample_X,sample_y)
        t_alpha = [1.94591015, 2.14179328, 2.48490665, 2.42209354, 3.1732565]
        assert pytest.approx(test_model.alpha, 0.01) == t_alpha, "Check the fit f
        unction"
```

```
In [8]: t_alpha
```

```
Out[8]: [1.94591015, 2.14179328, 2.48490665, 2.42209354, 3.1732565]
```

```
In [9]: test_model.alpha
```

```
Out[9]: array([1.94591015, 2.14179328, 2.48490665, 2.42209354, 3.17325658])
```

Use the fit function to fit the Adaboost classifier with 150 base decision tree stumps. [5 pts, Peer Review]

Note: Use data.x_train and data.y_train in fit

```
In [10]: # use fit function to fit Adaboost classifier called clf with 150 base de
         cision stumps
         # your code here

         clf = AdaBoost(n_learners=150, base=DecisionTreeClassifier(max_depth=1))
         clf.fit(data.x_train, data.y_train)
```

```
Out[10]: <__main__.AdaBoost at 0x76357694d990>
```

```
In [11]: # tests using the fit function to fit AdaBoost classifier with 150 base d
         ecision stumps
```

```
In [12]: # use fit function to fit Adaboost classifier called clf with 300 base de
         cision stumps
         # your code here

         clf = AdaBoost(n_learners=300, base=DecisionTreeClassifier(max_depth=1))
         clf.fit(data.x_train, data.y_train)

Out[12]: <__main__.AdaBoost at 0x763576907110>
```

```
In [13]: clf = AdaBoost(n_learners=300, base=DecisionTreeClassifier(max_depth=1))

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

         y_pred=clf.predict(data.x_test)

         pred_score = 1- clf.score(data.x_test, data.y_test)

         print('Misclassification error on test data: %0.3f' % pred_score)

         Misclassification error on test data: 0.040
```

Part B [5 pts]: After your `fit` method is working properly, implement the `predict` method to make predictions for unseen examples. You can test out the predictions in the cell below. **Note**: Remember that AdaBoost assumes that your predictions are of the form $y \in \{-1, 1\}$.

Just print out your predictions on the training set in the cell below.

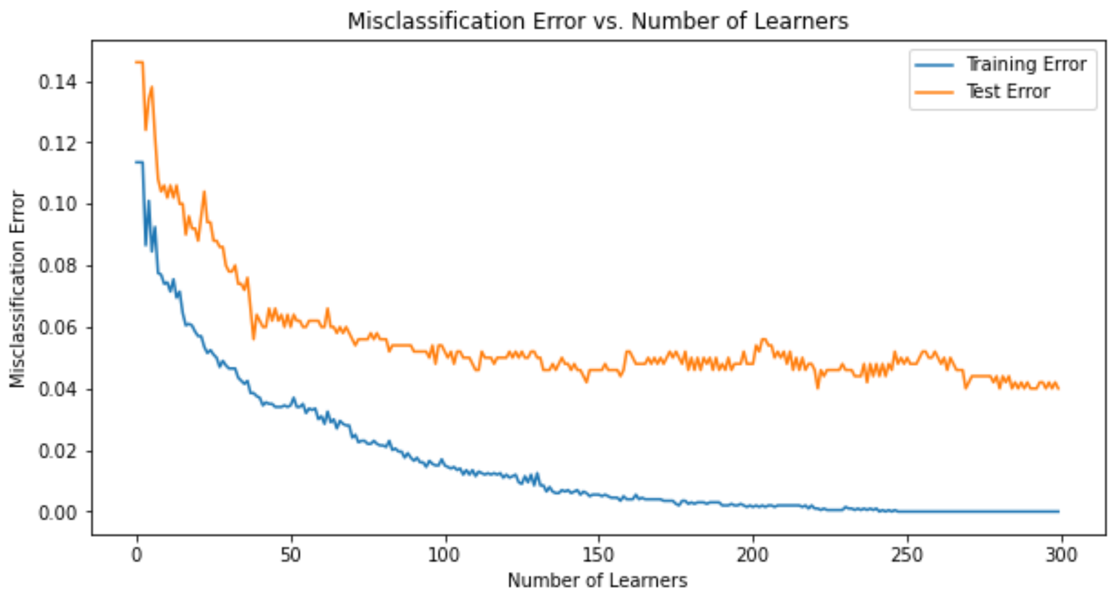
```
In [14]: # print out predictions on the training set
         # your code here

         train_predict = clf.predict(data.x_train)
         print(train_predict)

         [ 1.  1. -1. ...  1.  1. -1.]
```

```
In [15]: max(train_predict)

Out[15]: 1.0
```

```
In [16]: # tests train_predict which uses the predict method
```

Part C [Peer Review]: Once your `predict` function is written up, you need to test the scores on the function. To do this compute the scores on the prediction in the `score` function. Use the `score` function to then complete `staged_score` to collect the scores for every boosting iterations. Plot the misclassification error for train and test sets (misclassification error = 1- score).
Note: your code for this section may cause the Validate button to time out. If you want to run the Validate button prior to submitting, you could comment out the code in this section after completing the Peer Review.

In [17]:
```python
# plot misclassification error for train and test sets
# your code here
train_scores = clf.staged_score(data.x_train, data.y_train)
test_scores = clf.staged_score(data.x_test, data.y_test)

plt.figure(figsize=(10, 5))
plt.plot(1 - train_scores, label='Training Error')
plt.plot(1 - test_scores, label='Test Error')
plt.xlabel('Number of Learners')
plt.ylabel('Misclassification Error')
plt.title('Misclassification Error vs. Number of Learners')
plt.legend()
plt.show()
```



Problem 2 [5 pts, Peer Review] : Building an Random Forest Classifier to classify MNIST digits 3 and 8.

Remember that training the random forest algorithms involves the following steps:

```
 for k=1 to K:

     a) build kth tree of depth d

     b) Return the kth tree trained on the subset of dataset with the random feature
splits
```

Predicting the classification result on new data involves returning the majority vote by all the trees in the random forest.

Part A [5 points, Peer Review]: Complete the `create_tree` function to build a new tree trained on a subset of data. Within this function a decision tree classifier is built and trained on the subset of data with the subset of features. Answer the Peer Review question for this section.

In [18]:
```python
class RandomForest():

    def __init__(self, x, y, sample_sz, n_trees=200, n_features='sqrt', m
ax_depth=10, min_samples_leaf=5):
        """
        Create a new random forest classifier.

        Args:
            x : Input Feature vector
            y : Corresponding Labels
            sample_sz : Sample size
            n_trees : Number of trees to ensemble
            n_features : Method to select subset of features
            max_depth : Maximum depth of the trees in the ensemble
            min_sample_leaf : Minimum number of samples per leaf
        """
        np.random.seed(42)
        if n_features == 'sqrt':
            self.n_features = int(np.sqrt(x.shape[1]))
        elif n_features == 'log2':
            self.n_features = int(np.log2(x.shape[1]))
        else:
            self.n_features = n_features
        print(self.n_features, "sha: ",x.shape[1])
        self.features_set = []
        self.x, self.y, self.sample_sz, self.max_depth, self.min_samples_
leaf  = x, y, sample_sz, max_depth, min_samples_leaf
        self.trees = [self.create_tree(i) for i in range(n_trees)]

    def create_tree(self,i):
        """
        create a single decision tree classifier
        """


        idxs = np.random.permutation(len(self.y))[:self.sample_sz]
        idxs = np.asarray(idxs)

        f_idxs = np.random.permutation(self.x.shape[1])[:self.n_features]
        f_idxs = np.asarray(f_idxs)


        if i==0:
            self.features_set = np.array(f_idxs, ndmin=2)
        else:
            self.features_set = np.append(self.features_set, np.array(f_i
dxs,ndmin=2),axis=0)

        # TODO: build a decision tree classifier and train it with x and
y that is a subset of data (use idxs and f_idxs)

        # your code here

        # Build a decision tree classifier and train it with x and y that
is a subset of data (use idxs and f_idxs)


        clf = DecisionTreeClassifier(max_depth = self.max_depth, min_samp
les_leaf=self.min_samples_leaf)
```

```
        x, y = self.x[idxs][:,f_idxs], self.y[idxs]
        clf.fit(x,y)

        return clf

    def predict(self, x):

        # TODO: create a vector of predictions  and return
        # You will have to return the predictions of the final ensembles 
based on the individual trees' predicitons


        predict_list= [t.predict(x[:,self.features_set[i,:]]) for i, t in 
zip(range(self.features_set.shape[0]),self.trees) ]
        predict_array=np.array(predict_list).transpose()
        print(predict_array.shape)
        predict_minus1 = np.array((predict_array==-1).sum(axis=1)/predict
_array.shape[1])
        predict_ = predict_minus1
        predict_ = np.empty(predict_minus1.shape)
        predict_[(predict_minus1 > 0.5)] = -1
        predict_[(predict_minus1 <= 0.5)] = 1

        return predict_

    def score(self, X, y):

        # TODO: Compute the score using the predict function and true lab
els y
        from sklearn.metrics import accuracy_score
        y_pred = self.predict(X)
        return accuracy_score(y, y_pred)
```

In [19]:
```
# tests create_tree function
```

Part B [Peer Review]: In this part you will have to complete three steps:

1. Complete the `predict` function in RandomForest class so as to make predictions using just the features.
2. Finally complete the RandomForest class by completing the `score` function to compute the random forest model's accuracy on any dataset.
3. Build a random forest classifier and train it on the MNIST data to classify 3s and 8s in the cell below. Then see how the classifier performs on the test data by computing the misclassification error. (Remember: error = 1-score)
   Answer the Peer Review questions about this section.

In [20]:
```python
# TODO: build a random forest classifier and make predictions

# your code here

clf = RandomForest(data.x_train, data.y_train,int(np.round(data.x_train.s
hape[0]*0.7)) )

y_pred=clf.predict(data.x_test)

pred_score = 1- clf.score(data.x_test, data.y_test)

print('Misclassification error on test data: %0.3f' % pred_score)
```

```
21 sha:  441
(500, 200)
(500, 200)
Misclassification error on test data: 0.040
```

Both the AdaBoost and Random Forest classifiers have the same misclassification error on your test data, which is 0.040. This means that, based on the test data, both classifiers performed equally well in terms of their ability to correctly classify instances.

Here are a few reasons why you might see identical misclassification errors for these two different algorithms:

1. **Data Complexity**: The dataset might be of a size or complexity where both classifiers are equally effective. If the dataset is relatively simple or well-suited to both algorithms, they might end up with similar performance metrics.
2. **Hyperparameter Tuning**: It's possible that both algorithms were tuned to similar hyperparameter settings that optimized their performance. If both models were well-tuned, they could achieve similar results.
3. **Randomness in Training**: Both algorithms involve some degree of randomness in their training process. Random Forest uses random subsets of features and samples, while AdaBoost adjusts weights on misclassified samples iteratively. Depending on the randomness in your specific training runs, you might see similar performance.
4. **Evaluation Metric**: The misclassification error is just one way to evaluate classifier performance. Even though both classifiers have the same misclassification error, they might differ in other aspects like precision, recall, or computational efficiency.
5. **Model Overlap**: Depending on the nature of the data and the problem, the decision boundaries learned by AdaBoost and Random Forest might be similar or overlap, leading to comparable performance on the test set.

In practice, it's also important to consider other metrics and potentially additional validation strategies (like cross-validation) to get a more comprehensive understanding of model performance.

In [ ]:
```
```