# Lab4

April 13, 2020

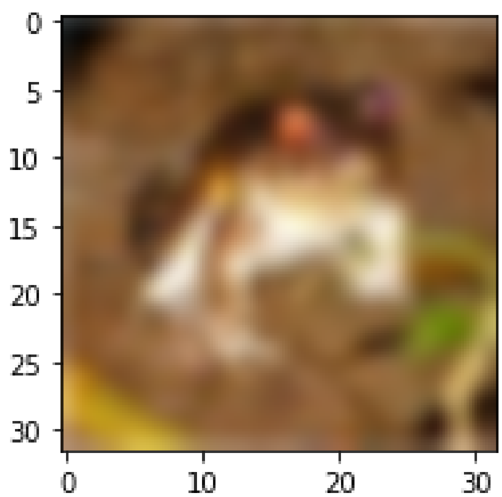
## 1 Problem 1

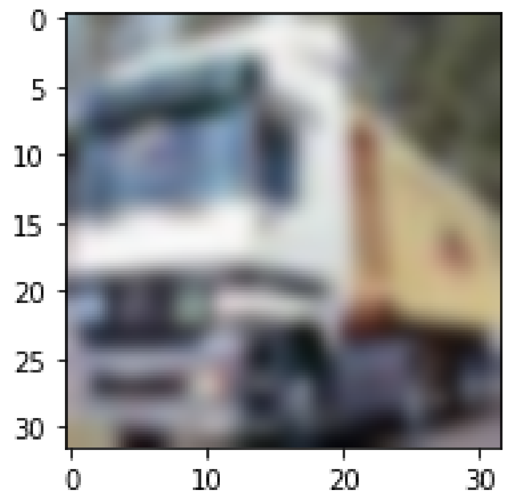
[2]:
```
from sklearn.datasets import fetch_openml
from matplotlib import pyplot
import numpy as np
```

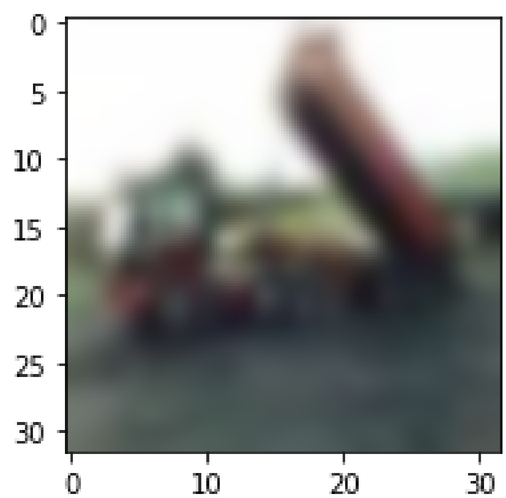
[3]:
```
data = fetch_openml(data_id=40926) # get CIFAR-10 data from OpenML
```

[3]:
```
def display_img(arr):
    R = arr[0:1024].reshape(32,32)/255.0
    G = arr[1024:2048].reshape(32,32)/255.0
    B = arr[2048:].reshape(32,32)/255.0

    img = np.dstack((R,G,B))
    fig = pyplot.figure(figsize=(3,3))
    ax = fig.add_subplot(111)
    ax.imshow(img,interpolation='bicubic')
```

[4]:
```
for i in range(0,3):
    display_img(data['data'][i])
```






[5]:
```
# Splitting the data up

from sklearn.model_selection import train_test_split

X = data['data']
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25)
```

[6]:
```
# Optimizing on a subset of training data for speed
from sklearn.preprocessing import StandardScaler

X_train = StandardScaler().fit_transform(X_train)
```

```python
X_test = StandardScaler().fit_transform(X_test)

X_train = X_train[:2000]
y_train = y_train[:2000]
```

[7]:
```python
# Let's train our logistic regression model now

from sklearn.linear_model import LogisticRegressionCV, LogisticRegression
from sklearn.metrics import roc_auc_score, accuracy_score

C_values = [1.5, 1.0, .75, .25, .1, .01, .001]


l1_ratio = 0.5  # L1 weight in the Elastic-Net regularization

# Set regularization parameter
for C in C_values:
    # turn down tolerance for short training time
    clf_l1_LR = LogisticRegression(C=C, multi_class='multinomial',␣
 ↪penalty='l1', tol=0.01, solver='saga')
    clf_l2_LR = LogisticRegression(C=C, multi_class='multinomial',␣
 ↪penalty='l2', tol=0.01, solver='saga')
    clf_en_LR = LogisticRegression(C=C, multi_class='multinomial',␣
 ↪penalty='elasticnet', solver='saga',
                                   l1_ratio=l1_ratio, tol=0.01)
    clf_l1_LR.fit(X_train, y_train)
    clf_l2_LR.fit(X_train, y_train)
    clf_en_LR.fit(X_train, y_train)

    coef_l1_LR = clf_l1_LR.coef_.ravel()
    coef_l2_LR = clf_l2_LR.coef_.ravel()
    coef_en_LR = clf_en_LR.coef_.ravel()

    # coef_l1_LR contains zeros due to the
    # L1 sparsity inducing norm

    sparsity_l1_LR = np.mean(coef_l1_LR == 0) * 100
    sparsity_l2_LR = np.mean(coef_l2_LR == 0) * 100
    sparsity_en_LR = np.mean(coef_en_LR == 0) * 100

    print("C=%.2f" % C)
    print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_l1_LR))
    print("{:<40} {:.2f}%".format("Sparsity with Elastic-Net penalty:
 ↪",sparsity_en_LR))
    print("{:<40} {:.2f}%".format("Sparsity with L2 penalty:", sparsity_l2_LR))
```

```
    print("{:<40} {:.2f}".format("Training Score with L1 penalty:", clf_l1_LR.
 ↪score(X_train, y_train)))
    print("{:<40} {:.2f}".format("Training Score with Elastic-Net penalty:",␣
 ↪clf_en_LR.score(X_train, y_train)))
    print("{:<40} {:.2f}".format("Training Score with L2 penalty:", clf_l2_LR.
 ↪score(X_train, y_train)))

    print("{:<40} {:.2f}".format("Testing Score with L1 penalty:", clf_l1_LR.
 ↪score(X_test, y_test)))
    print("{:<40} {:.2f}".format("Testing Score with Elastic-Net penalty:",␣
 ↪clf_en_LR.score(X_test, y_test)))
    print("{:<40} {:.2f}".format("Testing Score with L2 penalty:", clf_l2_LR.
 ↪score(X_test, y_test)))
```

```
C=1.50
Sparsity with L1 penalty:                3.16%
Sparsity with Elastic-Net penalty:       0.42%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.81
Training Score with Elastic-Net penalty: 0.82
Training Score with L2 penalty:          0.84
Testing Score with L1 penalty:           0.32
Testing Score with Elastic-Net penalty:  0.32
Testing Score with L2 penalty:           0.31
C=1.00
Sparsity with L1 penalty:                12.02%
Sparsity with Elastic-Net penalty:       2.67%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.79
Training Score with Elastic-Net penalty: 0.82
Training Score with L2 penalty:          0.84
Testing Score with L1 penalty:           0.32
Testing Score with Elastic-Net penalty:  0.32
Testing Score with L2 penalty:           0.31
C=0.75
Sparsity with L1 penalty:                24.73%
Sparsity with Elastic-Net penalty:       7.47%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.78
Training Score with Elastic-Net penalty: 0.81
Training Score with L2 penalty:          0.84
Testing Score with L1 penalty:           0.33
Testing Score with Elastic-Net penalty:  0.32
Testing Score with L2 penalty:           0.31
C=0.25
Sparsity with L1 penalty:                68.69%
Sparsity with Elastic-Net penalty:       39.90%
```

```
Sparsity with L2 penalty:                 0.00%
Training Score with L1 penalty:           0.67
Training Score with Elastic-Net penalty: 0.74
Training Score with L2 penalty:           0.84
Testing Score with L1 penalty:            0.34
Testing Score with Elastic-Net penalty:  0.33
Testing Score with L2 penalty:            0.32
C=0.10
Sparsity with L1 penalty:                 89.04%
Sparsity with Elastic-Net penalty:       74.51%
Sparsity with L2 penalty:                 0.00%
Training Score with L1 penalty:           0.54
Training Score with Elastic-Net penalty: 0.63
Training Score with L2 penalty:           0.83
Testing Score with L1 penalty:            0.35
Testing Score with Elastic-Net penalty:  0.35
Testing Score with L2 penalty:            0.32
C=0.01
Sparsity with L1 penalty:                 99.65%
Sparsity with Elastic-Net penalty:       98.29%
Sparsity with L2 penalty:                 0.00%
Training Score with L1 penalty:           0.23
Training Score with Elastic-Net penalty: 0.32
Training Score with L2 penalty:           0.74
Testing Score with L1 penalty:            0.19
Testing Score with Elastic-Net penalty:  0.28
Testing Score with L2 penalty:            0.33
C=0.00
Sparsity with L1 penalty:                 100.00%
Sparsity with Elastic-Net penalty:       100.00%
Sparsity with L2 penalty:                 0.00%
Training Score with L1 penalty:           0.11
Training Score with Elastic-Net penalty: 0.11
Training Score with L2 penalty:           0.55
Testing Score with L1 penalty:            0.10
Testing Score with Elastic-Net penalty:  0.10
Testing Score with L2 penalty:            0.35
```

By examining the above results, it seems that our models tend to break down somewhere between C=0.10 and C=0.01. The percentage of sparsity therefore starts affecting our model somewhere between 90-99% for L1, and 76-98% for elasticnet. Let's use these results to refine our parameters and find the optimal model.

[8]:
```
C_values = [.15, .14, .13, .12, .11, .1]

# Set regularization parameter
for C in C_values:
    # turn down tolerance for short training time
```

```
    clf_l1_LR = LogisticRegression(C=C, multi_class='multinomial',␣
↪penalty='l1', tol=0.01, solver='saga')
    clf_l2_LR = LogisticRegression(C=C, multi_class='multinomial',␣
↪penalty='l2', tol=0.01, solver='saga')
    clf_en_LR = LogisticRegression(C=C, multi_class='multinomial',␣
↪penalty='elasticnet', solver='saga',
                                   l1_ratio=l1_ratio, tol=0.01)
    clf_l1_LR.fit(X_train, y_train)
    clf_l2_LR.fit(X_train, y_train)
    clf_en_LR.fit(X_train, y_train)

    coef_l1_LR = clf_l1_LR.coef_.ravel()
    coef_l2_LR = clf_l2_LR.coef_.ravel()
    coef_en_LR = clf_en_LR.coef_.ravel()

    # coef_l1_LR contains zeros due to the
    # L1 sparsity inducing norm

    sparsity_l1_LR = np.mean(coef_l1_LR == 0) * 100
    sparsity_l2_LR = np.mean(coef_l2_LR == 0) * 100
    sparsity_en_LR = np.mean(coef_en_LR == 0) * 100

    print("C=%.2f" % C)
    print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_l1_LR))
    print("{:<40} {:.2f}%".format("Sparsity with Elastic-Net penalty:
↪",sparsity_en_LR))
    print("{:<40} {:.2f}%".format("Sparsity with L2 penalty:", sparsity_l2_LR))

    print("{:<40} {:.2f}".format("Training Score with L1 penalty:", clf_l1_LR.
↪score(X_train, y_train)))
    print("{:<40} {:.2f}".format("Training Score with Elastic-Net penalty:",␣
↪clf_en_LR.score(X_train, y_train)))
    print("{:<40} {:.2f}".format("Training Score with L2 penalty:", clf_l2_LR.
↪score(X_train, y_train)))

    print("{:<40} {:.2f}".format("Testing Score with L1 penalty:", clf_l1_LR.
↪score(X_test, y_test)))
    print("{:<40} {:.2f}".format("Testing Score with Elastic-Net penalty:",␣
↪clf_en_LR.score(X_test, y_test)))
    print("{:<40} {:.2f}".format("Testing Score with L2 penalty:", clf_l2_LR.
↪score(X_test, y_test)))
```

```
C=0.15
Sparsity with L1 penalty:                81.95%
Sparsity with Elastic-Net penalty:       62.65%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.58
```

```
Training Score with Elastic-Net penalty: 0.69
Training Score with L2 penalty:          0.83
Testing Score with L1 penalty:           0.35
Testing Score with Elastic-Net penalty:  0.34
Testing Score with L2 penalty:           0.32
C=0.14
Sparsity with L1 penalty:                82.36%
Sparsity with Elastic-Net penalty:       64.33%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.57
Training Score with Elastic-Net penalty: 0.68
Training Score with L2 penalty:          0.83
Testing Score with L1 penalty:           0.35
Testing Score with Elastic-Net penalty:  0.34
Testing Score with L2 penalty:           0.32
C=0.13
Sparsity with L1 penalty:                85.24%
Sparsity with Elastic-Net penalty:       61.95%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.56
Training Score with Elastic-Net penalty: 0.67
Training Score with L2 penalty:          0.83
Testing Score with L1 penalty:           0.35
Testing Score with Elastic-Net penalty:  0.34
Testing Score with L2 penalty:           0.32
C=0.12
Sparsity with L1 penalty:                86.34%
Sparsity with Elastic-Net penalty:       70.66%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.56
Training Score with Elastic-Net penalty: 0.66
Training Score with L2 penalty:          0.83
Testing Score with L1 penalty:           0.35
Testing Score with Elastic-Net penalty:  0.34
Testing Score with L2 penalty:           0.32
C=0.11
Sparsity with L1 penalty:                87.18%
Sparsity with Elastic-Net penalty:       72.00%
Sparsity with L2 penalty:                0.00%
Training Score with L1 penalty:          0.55
Training Score with Elastic-Net penalty: 0.65
Training Score with L2 penalty:          0.83
Testing Score with L1 penalty:           0.35
Testing Score with Elastic-Net penalty:  0.35
Testing Score with L2 penalty:           0.32
C=0.10
Sparsity with L1 penalty:                89.04%
Sparsity with Elastic-Net penalty:       73.38%
```

```
Sparsity with L2 penalty:                 0.00%
Training Score with L1 penalty:           0.54
Training Score with Elastic-Net penalty: 0.62
Training Score with L2 penalty:           0.82
Testing Score with L1 penalty:            0.35
Testing Score with Elastic-Net penalty:  0.35
Testing Score with L2 penalty:            0.32
```

We are getting pretty similar numbers for a lot of these models so we'll just pick the most consistent and general one to be our model (C=0.10, L1 penalty) shown above. With this optimal model we are getting a testing accuracy score of around 35%. Keep in mind that the model is trained on a subset of the training data to improve speed (training on all date would likely increase our models accuracy).

## 2 Problem 2

[4]:
```
data = fetch_openml(data_id=554) # get CIFAR-10 data from OpenML
```

[10]:
```
X = data['data']
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25)
```

[11]:
```
# Optimizing on a subset of training data
X_train = StandardScaler().fit_transform(X_train)
X_test = StandardScaler().fit_transform(X_test)

X_train = X_train[:2000]
y_train = y_train[:2000]
print(X_train.shape)
```

```
(2000, 784)
```

[14]:
```
l1_ratio = 0.5  # L1 weight in the Elastic-Net regularization

# Set regularization parameter
for C in [1.4, 1.2, 1.1, 0.1, 0.01]:
    # turn down tolerance for short training time
    clf_l1_LR = LogisticRegression(C=C, multi_class='multinomial',␣
 ↪penalty='l1', tol=0.01, solver='saga')
    clf_l2_LR = LogisticRegression(C=C, multi_class='multinomial',␣
 ↪penalty='l2', tol=0.01, solver='saga')
    clf_en_LR = LogisticRegression(C=C,␣
 ↪multi_class='multinomial',penalty='elasticnet', solver='saga',
                                   l1_ratio=l1_ratio, tol=0.01)
    clf_l1_LR.fit(X_train, y_train)
    clf_l2_LR.fit(X_train, y_train)
    clf_en_LR.fit(X_train, y_train)
```

```python
# coef_l1_LR contains zeros due to the
# L1 sparsity inducing norm

coef_l1_LR = clf_l1_LR.coef_.ravel()
coef_l2_LR = clf_l2_LR.coef_.ravel()
coef_en_LR = clf_en_LR.coef_.ravel()

sparsity_l1_LR = np.mean(coef_l1_LR == 0) * 100
sparsity_l2_LR = np.mean(coef_l2_LR == 0) * 100
sparsity_en_LR = np.mean(coef_en_LR == 0) * 100

print("C=%.2f" % C)
print("{:<40} {:.2f}%".format("Sparsity with L1 penalty:", sparsity_l1_LR))
print("{:<40} {:.2f}%".format("Sparsity with Elastic-Net penalty:",␣
↪sparsity_en_LR))
print("{:<40} {:.2f}%".format("Sparsity with L2 penalty:", sparsity_l2_LR))

print("{:<40} {:.2f}".format("Training Score with L1 penalty:",clf_l1_LR.
↪score(X_train, y_train)))
print("{:<40} {:.2f}".format("Training Score with Elastic-Net penalty:
↪",clf_en_LR.score(X_train, y_train)))
print("{:<40} {:.2f}".format("Training Score with L2 penalty:",clf_l2_LR.
↪score(X_train, y_train)))

print("{:<40} {:.2f}".format("Testing Score with L1 penalty:",clf_l1_LR.
↪score(X_test, y_test)))
print("{:<40} {:.2f}".format("Testing Score with Elastic-Net penalty:
↪",clf_en_LR.score(X_test, y_test)))
print("{:<40} {:.2f}".format("Testing Score with L2 penalty:",clf_l2_LR.
↪score(X_test, y_test)))
```

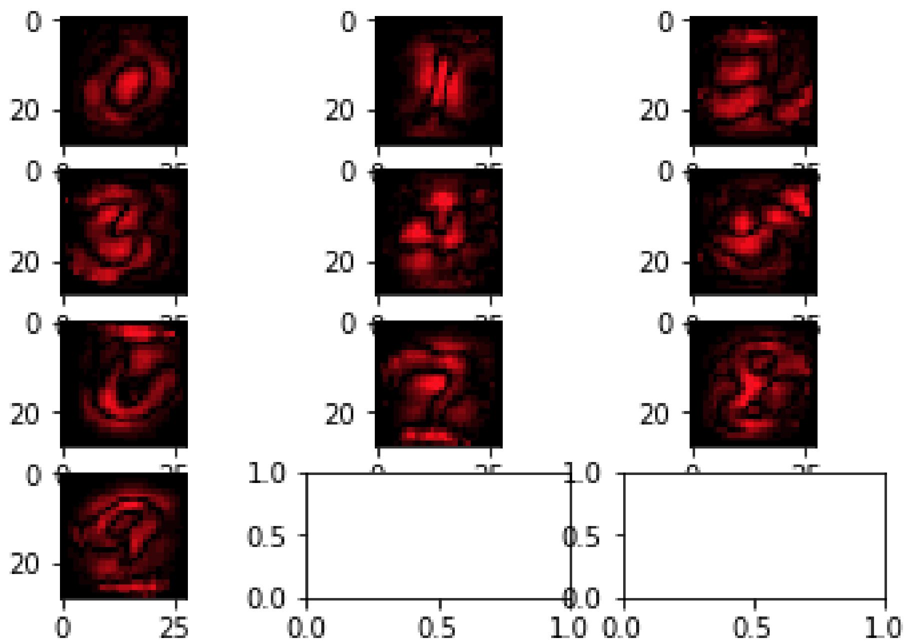
```
C=1.40
Sparsity with L1 penalty:                23.61%
Sparsity with Elastic-Net penalty:       20.10%
Sparsity with L2 penalty:                8.67%
Training Score with L1 penalty:          0.90
Training Score with Elastic-Net penalty: 0.90
Training Score with L2 penalty:          0.90
Testing Score with L1 penalty:           0.86
Testing Score with Elastic-Net penalty:  0.86
Testing Score with L2 penalty:           0.86
C=1.20
Sparsity with L1 penalty:                24.31%
Sparsity with Elastic-Net penalty:       20.70%
Sparsity with L2 penalty:                8.67%
Training Score with L1 penalty:          0.90
```

```
Training Score with Elastic-Net penalty: 0.90
Training Score with L2 penalty:          0.90
Testing Score with L1 penalty:           0.86
Testing Score with Elastic-Net penalty:  0.86
Testing Score with L2 penalty:           0.86
C=1.10
Sparsity with L1 penalty:                24.97%
Sparsity with Elastic-Net penalty:       21.68%
Sparsity with L2 penalty:                8.67%
Training Score with L1 penalty:          0.90
Training Score with Elastic-Net penalty: 0.90
Training Score with L2 penalty:          0.90
Testing Score with L1 penalty:           0.86
Testing Score with Elastic-Net penalty:  0.86
Testing Score with L2 penalty:           0.86
C=0.10
Sparsity with L1 penalty:                57.40%
Sparsity with Elastic-Net penalty:       43.18%
Sparsity with L2 penalty:                8.67%
Training Score with L1 penalty:          0.89
Training Score with Elastic-Net penalty: 0.89
Training Score with L2 penalty:          0.90
Testing Score with L1 penalty:           0.85
Testing Score with Elastic-Net penalty:  0.86
Testing Score with L2 penalty:           0.86
C=0.01
Sparsity with L1 penalty:                92.60%
Sparsity with Elastic-Net penalty:       83.65%
Sparsity with L2 penalty:                8.67%
Training Score with L1 penalty:          0.71
Training Score with Elastic-Net penalty: 0.82
Training Score with L2 penalty:          0.90
Testing Score with L1 penalty:           0.68
Testing Score with Elastic-Net penalty:  0.80
Testing Score with L2 penalty:           0.86
```

When we set C=1 and use L2 penalty we get the best and most general function. Thus, our optimal model is performing with a testing score of around 86%. Keep in mind that the model is trained on a subset of the training data to improve speed (training on all date would likely increase our models accuracy).

[13]:
```
opt_clf = LogisticRegression(C=.01, multi_class='multinomial', penalty='l2',␣
 ↪tol=0.01, solver='saga')
opt_clf.fit(X_train, y_train)

fig, axes = pyplot.subplots(4, 3)

for i, (axes_row) in enumerate(axes):
```

```
    if (i == 3):
      coefs = opt_clf.coef_[9]

      axes_row[0].imshow(np.abs(coefs.reshape(28,28)),␣
↪interpolation='nearest',
                          cmap='gist_stern', vmax=1, vmin=0)
    else:
      coefs = opt_clf.coef_[3*i:3*i+3]


      for ax, coef in zip(axes_row, coefs):
        ax.imshow(np.abs(coef.reshape(28, 28)), interpolation='nearest',
                  cmap='gist_stern', vmax=1, vmin=0)
```



# 3 Problem 3

[1]:
```
from sklearn.datasets import fetch_openml
import matplotlib.pyplot as plt
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
```

```
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV
from pprint import pprint

%matplotlib inline
```

[5]:
```
mnist_raw = fetch_openml(data_id=554) # get CIFAR-10 data from OpenML
```

[3]:
```
print(mnist_raw.keys())
```

```
dict_keys(['data', 'target', 'frame', 'feature_names', 'target_names', 'DESCR',
'details', 'categories', 'url'])
```

Data and target are clearly seperated which will make the proccess easier

[4]:
```
X_data = mnist_raw['data']
Y_target = mnist_raw['target']
```

X_data acts as our X with pixel intensities from 0 to 255 which are of 28 x 28 (784) images. Y_target acts as our y containing all labels from 0 - 9 mapping to X_data.

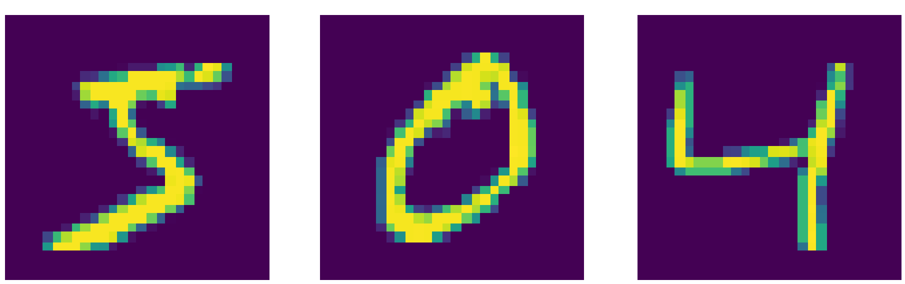
Y_target contains the integer values as strings as seen above. Let's convert it to integers

[5]:
```
Y_target = Y_target.astype(np.uint8)
Y_target[2]
```

[5]:
```
4
```

[6]:
```
for i in range(0, 3):
    digit = X_data[i] # represents first pixel number
    digit_pixels = digit.reshape(28, 28) # reshape to 28 x 28 matrix
    plt.subplot(131 + i) # subplot smaller than default
    plt.imshow(digit_pixels) # takes array image
    plt.axis('off')
    print(Y_target[i])
```

```
5
0
4
```



Split up Train and Test

[7]:
```
X_train, X_test, y_train, y_test = X_data[:60000], X_data[60000:], Y_target[:
↪60000], Y_target[60000:]
print('Train Data: ', X_train, '\n', 'Test Data:', X_test, '\n',
      'Train label: ', y_train, '\n', 'Test Label: ', y_test)
```

```
Train Data:  [[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]
 Test Data: [[0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 …
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]
 [0. 0. 0. … 0. 0. 0.]]
 Train label:  [5 0 4 … 5 6 8]
 Test Label:  [7 2 1 … 4 5 6]
```

### 3.0.1 Part 1

[31]:
```
# Baseline Random Forest
rfc_model = RandomForestClassifier(n_estimators=100,
                                    criterion='gini',
                                    max_depth=None,
                                    min_samples_split=2,
                                    min_samples_leaf=1,
                                    min_weight_fraction_leaf=0.0,
                                    max_features='auto', max_leaf_nodes=None,
                                    min_impurity_decrease=0.0,
                                    min_impurity_split=None, bootstrap=True,
                                    oob_score=False, n_jobs=None,
                                    random_state=42, verbose=0,
                                    warm_start=False,
                                    class_weight=None,
                                    ccp_alpha=0.0, max_samples=None
                                   )
```

[32]:
```
# Fit Baseline rfc
rfc_model.fit(X_train, y_train)
```

[32]:
```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=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=100,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

[10]:
```
# Check 3-fold Cross-Validation to see accuracy
# Have to make sure that scoring is set to accuracy so we can compare to␣
 ↪Logistic
cross_val_score(rfc_model, X_train, y_train, cv=3, scoring='accuracy')
```

[10]:
```
array([0.9646 , 0.96255, 0.9666 ])
```

3-Fold cross validation gets between 96-97% accuracy.

[61]:
```
# Check 5-fold Cross-Validation to see accuracy
cross_val_score(rfc_model, X_train, y_train, cv=5, scoring='accuracy')
```

[61]:
```
array([0.96858333, 0.96558333, 0.96366667, 0.96233333, 0.97016667])
```

5-Fold cross validation also gets between 96-97% accuracy

[63]:
```
rfc_score = rfc_model.score(X_test, y_test)
rfc_score
```

[63]:
```
0.9694
```

The actual score of this model is 0.9694.

Now we will use Cross Validation to get a rough estimate of good hyperparameters. Below are the current hyperparameters and we need to optimize them to get the highest possible accuracy.

[11]:
```
pprint(rfc_model.get_params())
```

```
{'bootstrap': True,
 'ccp_alpha': 0.0,
 'class_weight': None,
 'criterion': 'gini',
 'max_depth': None,
 'max_features': 'auto',
 'max_leaf_nodes': None,
 'max_samples': 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': 100,
 'n_jobs': None,
 'oob_score': False,
 'random_state': 42,
 'verbose': 0,
 'warm_start': False}
```

For this section, we will use RandomizedSearchCV to randomly try different sets of hyperparameters at random to narrow down where the best ones are. This differs from usual Grid Search as that is more fine-tuned and does not try random hyperparameters. The included hyperparameters are the ones we found most impactful to the model's score.

[29]:
```
# Number of trees in random forest
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
# Number of features to consider at every split
max_features = ['auto', 'sqrt']
# Maximum number of levels in tree
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
# Minimum number of samples required to split a node
min_samples_split = [2, 5, 10]
# Minimum number of samples required at each leaf node
min_samples_leaf = [1, 2, 4]
# Whether bootstrap samples are used when building trees
# If False, the whole datset is used to build each tree
bootstrap = [True, False]
# Create the random grid
random_grid = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
pprint(random_grid)
```

```
{'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 10],
 'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}
```

[30]:
```
# 5 Fold CV, Use all cores, 150 Combinations
rf = RandomForestClassifier();
rf_random = RandomizedSearchCV(estimator = rf,
                               param_distributions = random_grid,
                               n_iter = 40,
```

```
                                cv = 3,
                                verbose=2,
                                random_state=42,
                                n_jobs = -1)
```

[74]:
```
rf_random.fit(X_train, y_train)
```

```
Fitting 3 folds for each of 40 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed: 31.5min
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed: 150.9min finished
```

[74]:
```
RandomizedSearchCV(cv=3, error_score=nan,
                   estimator=RandomForestClassifier(bootstrap=True,
                                                    ccp_alpha=0.0,
                                                    class_weight=None,
                                                    criterion='gini',
                                                    max_depth=None,
                                                    max_features='auto',
                                                    max_leaf_nodes=None,
                                                    max_samples=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=100,
                                                    n_jobs…
                   param_distributions={'bootstrap': [True, False],
                                        'max_depth': [10, 20, 30, 40, 50, 60,
                                                      70, 80, 90, 100, 110,
                                                      None],
                                        'max_features': ['auto', 'sqrt'],
                                        'min_samples_leaf': [1, 2, 4],
                                        'min_samples_split': [2, 5, 10],
                                        'n_estimators': [200, 400, 600, 800,
                                                         1000, 1200, 1400, 1600,
                                                         1800, 2000]},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

[82]:
```
#Return best parameters from RandomSearchCV
rf_random.best_params_
```

[82]:
```
{'n_estimators': 1800,
 'min_samples_split': 2,
 'min_samples_leaf': 1,
```

```
 'max_features': 'auto',
 'max_depth': 50,
 'bootstrap': False}
```

[25]:
```
#made this one to prevent re-running RSCV
rfc_new = RandomForestClassifier(n_estimators= 1800,
                                 min_samples_split= 2,
                                 min_samples_leaf= 1,
                                 max_features= 'auto',
                                 max_depth= 50,
                                 bootstrap= False)
```

[26]:
```
rfc_new.fit(X_train, y_train)
```

[26]:
```
RandomForestClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=50, max_features='auto',
                       max_leaf_nodes=None, max_samples=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=1800,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

The above hyperparameters are the best from the RandomizedSearchCV.

[35]:
```
print("Base Score:")
print(base_score)
```

```
Base Score:
0.9705
```

[34]:
```
print("Tuned Score:")
print(random_score)
```

```
Tuned Score:
0.9751
```

[33]:
```
#Compare accuracy of tuned model vs base
random_score = rfc_new.score(X_test, y_test)
base_score = rfc_model.score(X_test, y_test)
print('Improvement of {:0.2f}%.'.format( 100 * (random_score - base_score) /␣
 ↪base_score))
```

```
Improvement of 0.47%.
```

Comparing our best model vs. logistic regression, got a best score of 0.9751 while logistic regression got scores from around 0.80 - 0.85. Hence, Random Forests appear to perform at a consistently higher accuracy.

### 3.0.2 Part 2

Now we will try Gradient Boosting to do the same thing: Get a base model, tune hyperparameters, compare tuned vs. base, compare best vs. RFC, compare best vs. logistic

[16]:
```
from xgboost.sklearn import XGBClassifier
import scipy.stats as st

one_to_left = st.beta(10, 1)
from_zero_positive = st.expon(0, 50)
gb_model = XGBClassifier(
    nthread=-1, #set to -1 to use all threads available
    n_jobs=-1, #set to -1 to use all threads, could be same as nthread
    seed=79, # random number seed
    random_state=0, #leave at 0 for reproducible results. Can try others (0)
    #set for GPU Hardware acceleration
    gpu_id=0,
    tree_method='gpu_hist'
    )
random_grid_gb = {
    "n_estimators": st.randint(3, 40),
    "max_depth": st.randint(3, 40),
    "learning_rate": st.uniform(0.05, 0.4),
    "colsample_bytree": one_to_left,
    "subsample": one_to_left,
    "gamma": st.uniform(0, 10),
    'reg_alpha': from_zero_positive,
    "min_child_weight": from_zero_positive,
}
```

[17]:
```
gb_model.fit(X_train, y_train)
```

[17]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=0,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=-1, nthread=-1, num_parallel_tree=1,
              objective='multi:softprob', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=None, seed=79, subsample=1,
              tree_method='gpu_hist', validate_parameters=False,
              verbosity=None)
```

[18]:
```
gb_base_score = gb_model.score(X_test, y_test)
gb_base_score
```

[18]:
```
0.9795
```

[21]: 
```python
cross_val_score(gb_model, X_train, y_train, cv=3, scoring='accuracy')
```

[21]: 
```
array([0.97555, 0.9716 , 0.97285])
```

The cross validation score (3 fold) ranges from 97 - 98 % accurate and the actual model scores 0.9795 on the test data.

[9]: 
```python
gb_random = RandomizedSearchCV(estimator = XGBClassifier(),
                               param_distributions = random_grid_gb,
                               n_iter = 20,
                               cv = 2,
                               verbose=2,
                               random_state=42,
                               n_jobs = 1)
```

[10]: 
```python
#find best one by fitting each random set of hp
gb_random.fit(X_train, y_train)
```

```
Fitting 2 folds for each of 20 candidates, totalling 40 fits
[CV] colsample_bytree=0.9252155845351104, gamma=1.5601864044243652,
learning_rate=0.11239780813448107, max_depth=13,
min_child_weight=30.73980825409684, n_estimators=38,
reg_alpha=7.708098373328053, subsample=0.9937572296628479

[Parallel(n_jobs=1)]: Using backend SequentialBackend with 1 concurrent workers.

[CV]  colsample_bytree=0.9252155845351104, gamma=1.5601864044243652,
learning_rate=0.11239780813448107, max_depth=13,
min_child_weight=30.73980825409684, n_estimators=38,
reg_alpha=7.708098373328053, subsample=0.9937572296628479, total= 6.8min
[CV] colsample_bytree=0.9252155845351104, gamma=1.5601864044243652,
learning_rate=0.11239780813448107, max_depth=13,
min_child_weight=30.73980825409684, n_estimators=38,
reg_alpha=7.708098373328053, subsample=0.9937572296628479

[Parallel(n_jobs=1)]: Done   1 out of   1 | elapsed:  6.8min remaining:    0.0s

[CV]  colsample_bytree=0.9252155845351104, gamma=1.5601864044243652,
learning_rate=0.11239780813448107, max_depth=13,
min_child_weight=30.73980825409684, n_estimators=38,
reg_alpha=7.708098373328053, subsample=0.9937572296628479, total= 2.0min
[CV] colsample_bytree=0.681939791143364, gamma=6.174815096271655,
learning_rate=0.29466126419531236, max_depth=27,
min_child_weight=17.211149627697075, n_estimators=30,
reg_alpha=182.01497768138802, subsample=0.9906270827741552
[CV]  colsample_bytree=0.681939791143364, gamma=6.174815096271655,
learning_rate=0.29466126419531236, max_depth=27,
min_child_weight=17.211149627697075, n_estimators=30,
reg_alpha=182.01497768138802, subsample=0.9906270827741552, total= 1.2min
[CV] colsample_bytree=0.681939791143364, gamma=6.174815096271655,
```

```
learning_rate=0.29466126419531236, max_depth=27,
min_child_weight=17.211149627697075, n_estimators=30,
reg_alpha=182.01497768138802, subsample=0.9906270827741552
[CV]  colsample_bytree=0.681939791143364, gamma=6.1748150962771655,
learning_rate=0.29466126419531236, max_depth=27,
min_child_weight=17.211149627697075, n_estimators=30,
reg_alpha=182.01497768138802, subsample=0.9906270827741552, total= 1.2min
[CV] colsample_bytree=0.9016525236942432, gamma=8.599404067363206,
learning_rate=0.3221230154351119, max_depth=11,
min_child_weight=3.363196543965212, n_estimators=6,
reg_alpha=142.53984394678014, subsample=0.9686996824842291
[CV]  colsample_bytree=0.9016525236942432, gamma=8.599404067363206,
learning_rate=0.3221230154351119, max_depth=11,
min_child_weight=3.363196543965212, n_estimators=6,
reg_alpha=142.53984394678014, subsample=0.9686996824842291, total=  24.4s
[CV] colsample_bytree=0.9016525236942432, gamma=8.599404067363206,
learning_rate=0.3221230154351119, max_depth=11,
min_child_weight=3.363196543965212, n_estimators=6,
reg_alpha=142.53984394678014, subsample=0.9686996824842291
[CV]  colsample_bytree=0.9016525236942432, gamma=8.599404067363206,
learning_rate=0.3221230154351119, max_depth=11,
min_child_weight=3.363196543965212, n_estimators=6,
reg_alpha=142.53984394678014, subsample=0.9686996824842291, total=  24.6s
[CV] colsample_bytree=0.8728775345487935, gamma=1.7336465350777208,
learning_rate=0.20642424302929635, max_depth=6,
min_child_weight=18.677329082881183, n_estimators=8,
reg_alpha=11.65601159428085, subsample=0.9952254582702151
[CV]  colsample_bytree=0.8728775345487935, gamma=1.7336465350777208,
learning_rate=0.20642424302929635, max_depth=6,
min_child_weight=18.677329082881183, n_estimators=8,
reg_alpha=11.65601159428085, subsample=0.9952254582702151, total=  20.8s
[CV] colsample_bytree=0.8728775345487935, gamma=1.7336465350777208,
learning_rate=0.20642424302929635, max_depth=6,
min_child_weight=18.677329082881183, n_estimators=8,
reg_alpha=11.65601159428085, subsample=0.9952254582702151
[CV]  colsample_bytree=0.8728775345487935, gamma=1.7336465350777208,
learning_rate=0.20642424302929635, max_depth=6,
min_child_weight=18.677329082881183, n_estimators=8,
reg_alpha=11.65601159428085, subsample=0.9952254582702151, total=  20.9s
[CV] colsample_bytree=0.777487224464793, gamma=7.272719958564209,
learning_rate=0.18061630752233415, max_depth=16,
min_child_weight=10.906734731563372, n_estimators=23,
reg_alpha=24.60651458971093, subsample=0.8873659886074203
[CV]  colsample_bytree=0.777487224464793, gamma=7.272719958564209,
learning_rate=0.18061630752233415, max_depth=16,
min_child_weight=10.906734731563372, n_estimators=23,
reg_alpha=24.60651458971093, subsample=0.8873659886074203, total= 1.2min
[CV] colsample_bytree=0.777487224464793, gamma=7.272719958564209,
```

```
learning_rate=0.18061630752233415, max_depth=16,
min_child_weight=10.906734731563372, n_estimators=23,
reg_alpha=24.60651458971093, subsample=0.8873659886074203
[CV]  colsample_bytree=0.777487224464793, gamma=7.272719958564209,
learning_rate=0.18061630752233415, max_depth=16,
min_child_weight=10.906734731563372, n_estimators=23,
reg_alpha=24.60651458971093, subsample=0.8873659886074203, total= 1.2min
[CV] colsample_bytree=0.9755279023629253, gamma=8.021969807540398,
learning_rate=0.07982025747190834, max_depth=9,
min_child_weight=27.53045308884588, n_estimators=3,
reg_alpha=11.076972025294298, subsample=0.809461138083018
[CV]  colsample_bytree=0.9755279023629253, gamma=8.021969807540398,
learning_rate=0.07982025747190834, max_depth=9,
min_child_weight=27.53045308884588, n_estimators=3,
reg_alpha=11.076972025294298, subsample=0.809461138083018, total=  11.6s
[CV] colsample_bytree=0.9755279023629253, gamma=8.021969807540398,
learning_rate=0.07982025747190834, max_depth=9,
min_child_weight=27.53045308884588, n_estimators=3,
reg_alpha=11.076972025294298, subsample=0.809461138083018
[CV]  colsample_bytree=0.9755279023629253, gamma=8.021969807540398,
learning_rate=0.07982025747190834, max_depth=9,
min_child_weight=27.53045308884588, n_estimators=3,
reg_alpha=11.076972025294298, subsample=0.809461138083018, total=  11.3s
[CV] colsample_bytree=0.9942230023599755, gamma=3.5846572854427263,
learning_rate=0.0963476238100519, max_depth=9,
min_child_weight=94.8688601614506, n_estimators=14, reg_alpha=20.09094006256505,
subsample=0.9718522760822342
[CV]  colsample_bytree=0.9942230023599755, gamma=3.5846572854427263,
learning_rate=0.0963476238100519, max_depth=9,
min_child_weight=94.8688601614506, n_estimators=14, reg_alpha=20.09094006256505,
subsample=0.9718522760822342, total=  38.1s
[CV] colsample_bytree=0.9942230023599755, gamma=3.5846572854427263,
learning_rate=0.0963476238100519, max_depth=9,
min_child_weight=94.8688601614506, n_estimators=14, reg_alpha=20.09094006256505,
subsample=0.9718522760822342
[CV]  colsample_bytree=0.9942230023599755, gamma=3.5846572854427263,
learning_rate=0.0963476238100519, max_depth=9,
min_child_weight=94.8688601614506, n_estimators=14, reg_alpha=20.09094006256505,
subsample=0.9718522760822342, total=  38.7s
[CV] colsample_bytree=0.8636115932647235, gamma=4.722149251619493,
learning_rate=0.09783769837532069, max_depth=16,
min_child_weight=94.49524523650217, n_estimators=7, reg_alpha=41.19437474429144,
subsample=0.9128850732864123
[CV]  colsample_bytree=0.8636115932647235, gamma=4.722149251619493,
learning_rate=0.09783769837532069, max_depth=16,
min_child_weight=94.49524523650217, n_estimators=7, reg_alpha=41.19437474429144,
subsample=0.9128850732864123, total=  18.5s
[CV] colsample_bytree=0.8636115932647235, gamma=4.722149251619493,
```

```
learning_rate=0.09783769837532069, max_depth=16,
min_child_weight=94.49524523650217, n_estimators=7, reg_alpha=41.19437474429144,
subsample=0.9128850732864123
[CV]  colsample_bytree=0.8636115932647235, gamma=4.722149251619493,
learning_rate=0.09783769837532069, max_depth=16,
min_child_weight=94.49524523650217, n_estimators=7, reg_alpha=41.19437474429144,
subsample=0.9128850732864123, total=  17.7s
[CV] colsample_bytree=0.9692206821226456, gamma=0.3142918568673425,
learning_rate=0.30456416450551216, max_depth=34,
min_child_weight=59.456851110666285, n_estimators=39,
reg_alpha=14.336941149164916, subsample=0.898377733005299
[CV]  colsample_bytree=0.9692206821226456, gamma=0.3142918568673425,
learning_rate=0.30456416450551216, max_depth=34,
min_child_weight=59.456851110666285, n_estimators=39,
reg_alpha=14.336941149164916, subsample=0.898377733005299, total= 1.6min
[CV] colsample_bytree=0.9692206821226456, gamma=0.3142918568673425,
learning_rate=0.30456416450551216, max_depth=34,
min_child_weight=59.456851110666285, n_estimators=39,
reg_alpha=14.336941149164916, subsample=0.898377733005299
[CV]  colsample_bytree=0.9692206821226456, gamma=0.3142918568673425,
learning_rate=0.30456416450551216, max_depth=34,
min_child_weight=59.456851110666285, n_estimators=39,
reg_alpha=14.336941149164916, subsample=0.898377733005299, total= 1.7min
[CV] colsample_bytree=0.8168821862783325, gamma=6.334037565104235,
learning_rate=0.3985842360750871, max_depth=30,
min_child_weight=12.323195802577699, n_estimators=22,
reg_alpha=111.54067026177805, subsample=0.8325253553053832
[CV]  colsample_bytree=0.8168821862783325, gamma=6.334037565104235,
learning_rate=0.3985842360750871, max_depth=30,
min_child_weight=12.323195802577699, n_estimators=22,
reg_alpha=111.54067026177805, subsample=0.8325253553053832, total= 1.1min
[CV] colsample_bytree=0.8168821862783325, gamma=6.334037565104235,
learning_rate=0.3985842360750871, max_depth=30,
min_child_weight=12.323195802577699, n_estimators=22,
reg_alpha=111.54067026177805, subsample=0.8325253553053832
[CV]  colsample_bytree=0.8168821862783325, gamma=6.334037565104235,
learning_rate=0.3985842360750871, max_depth=30,
min_child_weight=12.323195802577699, n_estimators=22,
reg_alpha=111.54067026177805, subsample=0.8325253553053832, total= 1.1min
[CV] colsample_bytree=0.9712419150915043, gamma=4.271077886262563,
learning_rate=0.37720590636899726, max_depth=5,
min_child_weight=0.34882045908253234, n_estimators=10,
reg_alpha=38.18807751704673, subsample=0.9099479985045033
[CV]  colsample_bytree=0.9712419150915043, gamma=4.271077886262563,
learning_rate=0.37720590636899726, max_depth=5,
min_child_weight=0.34882045908253234, n_estimators=10,
reg_alpha=38.18807751704673, subsample=0.9099479985045033, total=  24.7s
[CV] colsample_bytree=0.9712419150915043, gamma=4.271077886262563,
```

```
learning_rate=0.37720590636899726, max_depth=5,
min_child_weight=0.34882045908253234, n_estimators=10,
reg_alpha=38.18807751704673, subsample=0.9099479985045033
[CV]  colsample_bytree=0.9712419150915043, gamma=4.271077886262563,
learning_rate=0.37720590636899726, max_depth=5,
min_child_weight=0.34882045908253234, n_estimators=10,
reg_alpha=38.18807751704673, subsample=0.9099479985045033, total=  24.6s
[CV] colsample_bytree=0.966451867947173, gamma=5.581020020173412,
learning_rate=0.21153446842321633, max_depth=14,
min_child_weight=14.503064749610264, n_estimators=4,
reg_alpha=62.28673944235778, subsample=0.5464854199446646
[CV]  colsample_bytree=0.966451867947173, gamma=5.581020020173412,
learning_rate=0.21153446842321633, max_depth=14,
min_child_weight=14.503064749610264, n_estimators=4,
reg_alpha=62.28673944235778, subsample=0.5464854199446646, total=  13.7s
[CV] colsample_bytree=0.966451867947173, gamma=5.581020020173412,
learning_rate=0.21153446842321633, max_depth=14,
min_child_weight=14.503064749610264, n_estimators=4,
reg_alpha=62.28673944235778, subsample=0.5464854199446646
[CV]  colsample_bytree=0.966451867947173, gamma=5.581020020173412,
learning_rate=0.21153446842321633, max_depth=14,
min_child_weight=14.503064749610264, n_estimators=4,
reg_alpha=62.28673944235778, subsample=0.5464854199446646, total=  13.8s
[CV] colsample_bytree=0.8925500839956471, gamma=6.807054515547668,
learning_rate=0.26237383332685454, max_depth=28,
min_child_weight=212.20184913076298, n_estimators=34,
reg_alpha=44.90986104263074, subsample=0.953237899714839
[CV]  colsample_bytree=0.8925500839956471, gamma=6.807054515547668,
learning_rate=0.26237383332685454, max_depth=28,
min_child_weight=212.20184913076298, n_estimators=34,
reg_alpha=44.90986104263074, subsample=0.953237899714839, total= 1.1min
[CV] colsample_bytree=0.8925500839956471, gamma=6.807054515547668,
learning_rate=0.26237383332685454, max_depth=28,
min_child_weight=212.20184913076298, n_estimators=34,
reg_alpha=44.90986104263074, subsample=0.953237899714839
[CV]  colsample_bytree=0.8925500839956471, gamma=6.807054515547668,
learning_rate=0.26237383332685454, max_depth=28,
min_child_weight=212.20184913076298, n_estimators=34,
reg_alpha=44.90986104263074, subsample=0.953237899714839, total= 1.1min
[CV] colsample_bytree=0.747093280352113, gamma=3.9882444244455306,
learning_rate=0.3765727492877536, max_depth=18,
min_child_weight=90.18223971231485, n_estimators=8, reg_alpha=59.50559999737251,
subsample=0.952251493220607
[CV]  colsample_bytree=0.747093280352113, gamma=3.9882444244455306,
learning_rate=0.3765727492877536, max_depth=18,
min_child_weight=90.18223971231485, n_estimators=8, reg_alpha=59.50559999737251,
subsample=0.952251493220607, total=  18.6s
[CV] colsample_bytree=0.747093280352113, gamma=3.9882444244455306,
```

```
learning_rate=0.3765727492877536, max_depth=18,
min_child_weight=90.18223971231485, n_estimators=8, reg_alpha=59.50559999737251,
subsample=0.952251493220607
[CV]  colsample_bytree=0.747093280352113, gamma=3.9882444244455306,
learning_rate=0.3765727492877536, max_depth=18,
min_child_weight=90.18223971231485, n_estimators=8, reg_alpha=59.50559999737251,
subsample=0.952251493220607, total=  18.0s
[CV] colsample_bytree=0.965713632659158, gamma=0.9617655109142076,
learning_rate=0.4262093057958416, max_depth=20,
min_child_weight=90.91855366176486, n_estimators=13,
reg_alpha=20.856621699758588, subsample=0.730215002586014
[CV]  colsample_bytree=0.965713632659158, gamma=0.9617655109142076,
learning_rate=0.4262093057958416, max_depth=20,
min_child_weight=90.91855366176486, n_estimators=13,
reg_alpha=20.856621699758588, subsample=0.730215002586014, total=  31.5s
[CV] colsample_bytree=0.965713632659158, gamma=0.9617655109142076,
learning_rate=0.4262093057958416, max_depth=20,
min_child_weight=90.91855366176486, n_estimators=13,
reg_alpha=20.856621699758588, subsample=0.730215002586014
[CV]  colsample_bytree=0.965713632659158, gamma=0.9617655109142076,
learning_rate=0.4262093057958416, max_depth=20,
min_child_weight=90.91855366176486, n_estimators=13,
reg_alpha=20.856621699758588, subsample=0.730215002586014, total=  32.2s
[CV] colsample_bytree=0.8349833924899642, gamma=5.5520081159946235,
learning_rate=0.2618602313424026, max_depth=4,
min_child_weight=279.34932640625937, n_estimators=14,
reg_alpha=40.85547697931888, subsample=0.9821412119610942
[CV]  colsample_bytree=0.8349833924899642, gamma=5.5520081159946235,
learning_rate=0.2618602313424026, max_depth=4,
min_child_weight=279.34932640625937, n_estimators=14,
reg_alpha=40.85547697931888, subsample=0.9821412119610942, total=  25.0s
[CV] colsample_bytree=0.8349833924899642, gamma=5.5520081159946235,
learning_rate=0.2618602313424026, max_depth=4,
min_child_weight=279.34932640625937, n_estimators=14,
reg_alpha=40.85547697931888, subsample=0.9821412119610942
[CV]  colsample_bytree=0.8349833924899642, gamma=5.5520081159946235,
learning_rate=0.2618602313424026, max_depth=4,
min_child_weight=279.34932640625937, n_estimators=14,
reg_alpha=40.85547697931888, subsample=0.9821412119610942, total=  25.1s
[CV] colsample_bytree=0.8700259294978819, gamma=4.045081271221901,
learning_rate=0.40510803950438395, max_depth=13,
min_child_weight=137.15925910424536, n_estimators=38,
reg_alpha=114.41152500887529, subsample=0.9986268536106073
[CV]  colsample_bytree=0.8700259294978819, gamma=4.045081271221901,
learning_rate=0.40510803950438395, max_depth=13,
min_child_weight=137.15925910424536, n_estimators=38,
reg_alpha=114.41152500887529, subsample=0.9986268536106073, total= 1.2min
[CV] colsample_bytree=0.8700259294978819, gamma=4.045081271221901,
```

```
learning_rate=0.40510803950438395, max_depth=13,
min_child_weight=137.15925910424536, n_estimators=38,
reg_alpha=114.41152500887529, subsample=0.9986268536106073
[CV]  colsample_bytree=0.8700259294978819, gamma=4.045081271221901,
learning_rate=0.40510803950438395, max_depth=13,
min_child_weight=137.15925910424536, n_estimators=38,
reg_alpha=114.41152500887529, subsample=0.9986268536106073, total= 1.2min
[CV] colsample_bytree=0.9837401231719675, gamma=5.487337893665861,
learning_rate=0.3267580790770773, max_depth=18,
min_child_weight=259.76514352957037, n_estimators=38,
reg_alpha=62.27086444492694, subsample=0.9508266748114431
[CV]  colsample_bytree=0.9837401231719675, gamma=5.487337893665861,
learning_rate=0.3267580790770773, max_depth=18,
min_child_weight=259.76514352957037, n_estimators=38,
reg_alpha=62.27086444492694, subsample=0.9508266748114431, total= 1.3min
[CV] colsample_bytree=0.9837401231719675, gamma=5.487337893665861,
learning_rate=0.3267580790770773, max_depth=18,
min_child_weight=259.76514352957037, n_estimators=38,
reg_alpha=62.27086444492694, subsample=0.9508266748114431
[CV]  colsample_bytree=0.9837401231719675, gamma=5.487337893665861,
learning_rate=0.3267580790770773, max_depth=18,
min_child_weight=259.76514352957037, n_estimators=38,
reg_alpha=62.27086444492694, subsample=0.9508266748114431, total= 1.3min
[CV] colsample_bytree=0.9196270427281408, gamma=5.683086033354716,
learning_rate=0.08746990713123699, max_depth=24,
min_child_weight=15.408007392941558, n_estimators=32,
reg_alpha=180.61547030814629, subsample=0.8642391331808185
[CV]  colsample_bytree=0.9196270427281408, gamma=5.683086033354716,
learning_rate=0.08746990713123699, max_depth=24,
min_child_weight=15.408007392941558, n_estimators=32,
reg_alpha=180.61547030814629, subsample=0.8642391331808185, total= 1.8min
[CV] colsample_bytree=0.9196270427281408, gamma=5.683086033354716,
learning_rate=0.08746990713123699, max_depth=24,
min_child_weight=15.408007392941558, n_estimators=32,
reg_alpha=180.61547030814629, subsample=0.8642391331808185
[CV]  colsample_bytree=0.9196270427281408, gamma=5.683086033354716,
learning_rate=0.08746990713123699, max_depth=24,
min_child_weight=15.408007392941558, n_estimators=32,
reg_alpha=180.61547030814629, subsample=0.8642391331808185, total= 1.8min
[CV] colsample_bytree=0.9418638777900282, gamma=4.925176938188639,
learning_rate=0.1280971951192178, max_depth=21,
min_child_weight=42.43209473424399, n_estimators=19,
reg_alpha=260.61166994910684, subsample=0.9721308467038826
[CV]  colsample_bytree=0.9418638777900282, gamma=4.925176938188639,
learning_rate=0.1280971951192178, max_depth=21,
min_child_weight=42.43209473424399, n_estimators=19,
reg_alpha=260.61166994910684, subsample=0.9721308467038826, total=  52.7s
[CV] colsample_bytree=0.9418638777900282, gamma=4.925176938188639,
```

```
learning_rate=0.1280971951192178, max_depth=21,
min_child_weight=42.43209473424399, n_estimators=19,
reg_alpha=260.61166994910684, subsample=0.9721308467038826
[CV]  colsample_bytree=0.9418638777900282, gamma=4.925176938188639,
learning_rate=0.1280971951192178, max_depth=21,
min_child_weight=42.43209473424399, n_estimators=19,
reg_alpha=260.61166994910684, subsample=0.9721308467038826, total=  54.9s

[Parallel(n_jobs=1)]: Done  40 out of  40 | elapsed: 39.3min finished
```

[10]:
```
RandomizedSearchCV(cv=2, error_score=nan,
                   estimator=XGBClassifier(base_score=None, booster=None,
                                           colsample_bylevel=None,
                                           colsample_bynode=None,
                                           colsample_bytree=None, gamma=None,
                                           gpu_id=None, importance_type='gain',
                                           interaction_constraints=None,
                                           learning_rate=None,
                                           max_delta_step=None, max_depth=None,
                                           min_child_weight=None, missing=nan,
                                           monotone_constraints=None,
                                           n…
                                        'n_estimators':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x000001F281841948>,
                                        'reg_alpha':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x000001F2E2434248>,
                                        'subsample':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x000001F2E2434208>},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

[11]:
```
#Return best parameters from RandomSearchCV
gb_random.best_params_
```

[11]:
```
{'colsample_bytree': 0.9692206821226456,
 'gamma': 0.3142918568673425,
 'learning_rate': 0.30456416450551216,
 'max_depth': 34,
 'min_child_weight': 59.456851110666285,
 'n_estimators': 39,
 'reg_alpha': 14.336941149164916,
 'subsample': 0.898377733005299}
```

[12]:
```
gb_new = XGBClassifier(colsample_bytree= 0.9692206821226456,
         gamma = 0.3142918568673425,
         learning_rate= 0.30456416450551216,
         max_depth= 34,
         min_child_weight=59.456851110666285,
```

```
          n_estimators=39,
          reg_alpha=14.336941149164916,
          subsample= 0.898377733005299)
```

[13]:
```
gb_new.fit(X_train, y_train)
```

[13]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.9692206821226456,
              gamma=0.3142918568673425, gpu_id=-1, importance_type='gain',
              interaction_constraints=None, learning_rate=0.30456416450551216,
              max_delta_step=0, max_depth=34,
              min_child_weight=59.456851110666285, missing=nan,
              monotone_constraints=None, n_estimators=39, n_jobs=0,
              num_parallel_tree=1, objective='multi:softprob', random_state=0,
              reg_alpha=14.336941149164916, reg_lambda=1, scale_pos_weight=None,
              subsample=0.898377733005299, tree_method=None,
              validate_parameters=False, verbosity=None)
```

[22]:
```
gb_random_score = gb_new.score(X_test, y_test)
gb_random_score
```

[22]:
```
0.9596
```

[37]:
```
#Compare accuracy of tuned XGB model vs base XGB
gb_random_score = gb_new.score(X_test, y_test)
print('Decrease of {:0.2f}%.'.format( 100 * (gb_random_score - gb_base_score) /
 ↪gb_base_score))
```

```
Decrease of -2.03%.
```

[21]:
```
#Compre accuracy of best XBG vs. tuned RFC
print('Increase of {:0.2f}%.'.format( 100 * (gb_base_score - random_score) /
 ↪random_score))
```

```
Increase of 7.24%.
```

Based on the results, the tuned Gradient Boosting model performed worse than the baseline Gradient Boosting and the Random Forest (tuned). This is probably because RanddomSearchCV is not all-inclusive and could likely find combinations that are not accurate. Also, some hyperparameters like reg_alpha go outside of the recommended values for an XGB model. The baseline XGB model performs better tan the tuned RFC as well.

Comparing with Random Forests and Logistic, Gradient Boosting with our best parameters got 0.9795 which is only slightly higher than Random Forest's 0.9751. It is tough to say that Gradient Boosting is definitively more accurate with the data provided. However, both of these models outperformed Logistic Regression.

# 4 Problem 4

[37]:
```
from sklearn.datasets import fetch_openml
from matplotlib import pyplot
from sklearn.metrics import roc_auc_score, accuracy_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import train_test_split
from sklearn.model_selection import RandomizedSearchCV
from pprint import pprint
import numpy as np
```

[6]:
```
data = fetch_openml(data_id=40926) # CIFAR-10 data from OpenML
```

[3]:
```
X = data['data']
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.25)
```

### 4.0.1 Part 1

[4]:
```
rfc_modelCF = RandomForestClassifier(n_estimators=100,
                                     criterion='gini',
                                     max_depth=None,
                                     min_samples_split=2,
                                     min_samples_leaf=1,
                                     min_weight_fraction_leaf=0.0,
                                     max_features='auto', max_leaf_nodes=None,
                                     min_impurity_decrease=0.0,
                                     min_impurity_split=None, bootstrap=True,
                                     oob_score=False, n_jobs=None,
                                     random_state=42, verbose=0,
                                     warm_start=False,
                                     class_weight=None,
                                     ccp_alpha=0.0, max_samples=None
                                    )
```

[5]:
```
rfc_modelCF.fit(X_train, y_train)
rfc_modelCF_preds = rfc_modelCF.predict(X_test)
print("Accuracy Score: {}".format(accuracy_score(y_test, rfc_modelCF_preds )))
```

```
Accuracy Score: 0.4362
```

[6]:
```
cross_val_score(rfc_modelCF, X_train, y_train, cv=5, scoring='accuracy')
```

[6]:
```
array([0.42833333, 0.42433333, 0.43633333, 0.41433333, 0.417     ])
```

[9]:
```
n_estimators = [int(x) for x in np.linspace(start = 200, stop = 2000, num = 10)]
max_features = ['auto', 'sqrt']
max_depth = [int(x) for x in np.linspace(10, 110, num = 11)]
max_depth.append(None)
min_samples_split = [2, 5, 10]
min_samples_leaf = [1, 2, 4]
bootstrap = [True, False]
random_gridCF = {'n_estimators': n_estimators,
               'max_features': max_features,
               'max_depth': max_depth,
               'min_samples_split': min_samples_split,
               'min_samples_leaf': min_samples_leaf,
               'bootstrap': bootstrap}
pprint(random_gridCF)
```

```
{'bootstrap': [True, False],
 'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, None],
 'max_features': ['auto', 'sqrt'],
 'min_samples_leaf': [1, 2, 4],
 'min_samples_split': [2, 5, 10],
 'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}
```

[11]:
```
rf_randomCF = RandomizedSearchCV(estimator = RandomForestClassifier(),
                                param_distributions = random_gridCF,
                                n_iter = 20,
                                cv = 2,
                                verbose=2,
                                random_state=42,
                                n_jobs = -1)
```

[12]:
```
rf_randomCF.fit(X_train, y_train)
```

```
Fitting 2 folds for each of 20 candidates, totalling 40 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed: 21.0min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed: 38.1min finished
```

[12]:
```
RandomizedSearchCV(cv=2, error_score=nan,
                   estimator=RandomForestClassifier(bootstrap=True,
                                                    ccp_alpha=0.0,
                                                    class_weight=None,
                                                    criterion='gini',
                                                    max_depth=None,
                                                    max_features='auto',
                                                    max_leaf_nodes=None,
                                                    max_samples=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=100,
                                                    n_jobs…
                   param_distributions={'bootstrap': [True, False],
                                        'max_depth': [10, 20, 30, 40, 50, 60,
                                                      70, 80, 90, 100, 110,
                                                      None],
                                        'max_features': ['auto', 'sqrt'],
                                        'min_samples_leaf': [1, 2, 4],
                                        'min_samples_split': [2, 5, 10],
                                        'n_estimators': [200, 400, 600, 800,
                                                         1000, 1200, 1400, 1600,
                                                         1800, 2000]},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

[13]:
```
#Return best parameters from RandomSearchCV
rf_randomCF.best_params_
```

[13]:
```
{'n_estimators': 2000,
 'min_samples_split': 2,
 'min_samples_leaf': 2,
 'max_features': 'auto',
 'max_depth': 50,
 'bootstrap': False}
```

[6]:
```
rfc_modelCFNew = RandomForestClassifier(n_estimators= 2000,
     min_samples_split=2,
     min_samples_leaf= 2,
     max_features='auto',
     max_depth= 50,
     bootstrap= False)
```

[7]:
```
rfc_modelCFNew.fit(X_train, y_train)
```

[7]:
```
RandomForestClassifier(bootstrap=False, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=50, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=2, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=2000,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

[8]:
```
#Compare accuracy of tuned model vs base
random_score = rfc_modelCFNew.score(X_test, y_test)
base_score = rfc_modelCF.score(X_test, y_test)
print('Tuned Improvement of {:0.2f}%.'.format( 100 * (random_score -␣
↪base_score) / base_score))
```

```
Tuned Improvement of 7.61%.
```

[21]:
```
print("Base Score:")
print(base_score)
```

```
Base Score:
0.419
```

[20]:
```
print("Tuned Score:")
print(random_score)
```

```
Tuned Score:
0.467
```

This time when we compare against logistic regression we get more interesting results. Logistic Regression scored around 0.8 on testing; however, it only scored 0.35 on the training data. With RFC, the score is significantly higher at 0.467 but it is still lower 50% accurate.

### 4.0.2 Part 2

[9]:
```
from xgboost.sklearn import XGBClassifier
import scipy.stats as st

one_to_left = st.beta(10, 1)
from_zero_positive = st.expon(0, 50)
gb_modelCF = XGBClassifier(
    nthread=-1, #set to -1 to use all threads available
    n_jobs=-1, #set to -1 to use all threads, could be same as nthread
    seed=79, # random number seed
    random_state=0, #leave at 0 for reproducible results. Can try others (0)
    #set for GPU Hardware acceleration
    gpu_id=0,
    tree_method='gpu_hist'
    )
random_grid_gbCF = {
    "n_estimators": st.randint(3, 40),
    "max_depth": st.randint(3, 15),
    "learning_rate": st.uniform(0.05, 0.3),
    "colsample_bytree": one_to_left,
    "subsample": one_to_left,
    "gamma": st.uniform(0, 10),
    'reg_alpha': from_zero_positive,
```

```
    "min_child_weight": st.uniform(1, 7)
}
```

[10]:
```
gb_modelCF.fit(X_train, y_train)
```

[10]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=0,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=-1, nthread=-1, num_parallel_tree=1,
              objective='multi:softprob', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=None, seed=79, subsample=1,
              tree_method='gpu_hist', validate_parameters=False,
              verbosity=None)
```

[16]:
```
gb_base_score = gb_modelCF.score(X_test, y_test)
gb_base_score
```

[16]:
```
0.5034
```

[10]:
```
cross_val_score(gb_model, X_train, y_train, cv=3, scoring='accuracy')
```

[10]:
```
array([0.4794, 0.4696, 0.476 ])
```

[11]:
```
gb_randomCF = RandomizedSearchCV(estimator = XGBClassifier(),
                                 param_distributions = random_grid_gbCF,
                                 n_iter = 20,
                                 cv = 2,
                                 verbose=2,
                                 random_state=42,
                                 n_jobs = -1)
```

[12]:
```
#find best one by fitting each random set of hp
gb_randomCF.fit(X_train, y_train)
```

```
Fitting 2 folds for each of 20 candidates, totalling 40 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 8 concurrent workers.
[Parallel(n_jobs=-1)]: Done  25 tasks      | elapsed: 36.9min
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed: 52.2min finished
```

[12]:
```
RandomizedSearchCV(cv=2, error_score=nan,
                   estimator=XGBClassifier(base_score=None, booster=None,
                                           colsample_bylevel=None,
                                           colsample_bynode=None,
                                           colsample_bytree=None, gamma=None,
                                           gpu_id=None, importance_type='gain',
```

```
                                                interaction_constraints=None,
                                                learning_rate=None,
                                                max_delta_step=None, max_depth=None,
                                                min_child_weight=None, missing=nan,
                                                monotone_constraints=None,
                                                n…
                                             'n_estimators':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x0000025ABF738CC8>,
                                             'reg_alpha':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x0000025ABF738048>,
                                             'subsample':
<scipy.stats._distn_infrastructure.rv_frozen object at 0x0000025ABF738088>},
                   pre_dispatch='2*n_jobs', random_state=42, refit=True,
                   return_train_score=False, scoring=None, verbose=2)
```

[13]:

```python
#Return best parameters from RandomSearchCV
gb_randomCF.best_params_
```

[13]:

```
{'colsample_bytree': 0.931729684144507,
 'gamma': 1.2203823484477883,
 'learning_rate': 0.19855307303338104,
 'max_depth': 5,
 'min_child_weight': 7.365242814551475,
 'n_estimators': 38,
 'reg_alpha': 10.05907999573974,
 'subsample': 0.9407313857835466}
```

[11]:

```python
gb_modelCFNew = XGBClassifier(
    nthread=-1, #set to -1 to use all threads available
    n_jobs=-1, #set to -1 to use all threads, could be same as nthread
    seed=79, # random number seed
    random_state=0, #leave at 0 for reproducible results. Can try others (0)
    #set for GPU Hardware acceleration
    gpu_id=0,
    tree_method='gpu_hist',
    colsample_bytree=0.931729684144507,
    gamma=1.2203823484477883,
    learning_rate=0.19855307303338104,
    max_depth=5,
    min_child_weight=7.365242814551475,
    n_estimators=38,
    reg_alpha=10.05907999573974,
    subsample=0.9407313857835466
    )
```

[12]:

```python
gb_modelCFNew.fit(X_train, y_train)
```

[12]:
```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=0.931729684144507,
              gamma=1.2203823484477883, gpu_id=0, importance_type='gain',
              interaction_constraints=None, learning_rate=0.19855307303338104,
              max_delta_step=0, max_depth=5, min_child_weight=7.365242814551475,
              missing=nan, monotone_constraints=None, n_estimators=38,
              n_jobs=-1, nthread=-1, num_parallel_tree=1,
              objective='multi:softprob', random_state=0,
              reg_alpha=10.05907999573974, reg_lambda=1, scale_pos_weight=None,
              seed=79, subsample=0.9407313857835466, tree_method='gpu_hist',
              validate_parameters=False, verbosity=None)
```

[13]:
```
#Compare accuracy of tuned XGB Model vs. base XGB Model
gb_random_score = gb_modelCFNew.score(X_test, y_test)
```

[21]:
```
print("Base:")
print(gb_base_score)
print("Random Search CV:")
print(gb_random_score)
print('Decrease of {:0.2f}%.'.format( 100 * (gb_random_score - gb_base_score) /␣
↪gb_base_score))
```

```
Base:
0.502
Random Search CV:
0.464
Decrease of -7.57%.
```

For the same reasons discussed from the last question, tuning XGB in this method will not necessarily increase the score accuracy. However, it is important to note that the base XGB model performs better than the Tuned RFC like in problem 3.

[18]:
```
#Compare accuracy of best XGB Model vs. Tuned RFC
print('Increase of {:0.2f}%.'.format( 100 * (gb_base_score - random_score) /␣
↪random_score))
```

```
Increase of 7.24%.
```

Now we will try one more model as per document specifications of variety. We will go with CatBoostClassifier.

[24]:
```
from catboost import Pool, CatBoostClassifier
```

[29]:
```
cat = CatBoostClassifier(iterations=10,
                         learning_rate=1,
                         depth=2,
                         loss_function='MultiClass')
```

[33]:
```python
train_dataset = Pool(data = X_train,
                     label = y_train)

eval_dataset = Pool(data=X_test,
                    label=y_test)
```

[34]:
```python
cat.fit(train_dataset)
```

```
0:      learn: 2.1748108        total: 304ms    remaining: 2.73s
1:      learn: 2.1165645        total: 542ms    remaining: 2.17s
2:      learn: 2.0765038        total: 781ms    remaining: 1.82s
3:      learn: 2.0497507        total: 998ms    remaining: 1.5s
4:      learn: 2.0318204        total: 1.22s    remaining: 1.22s
5:      learn: 2.0125454        total: 1.46s    remaining: 975ms
6:      learn: 1.9915536        total: 1.73s    remaining: 742ms
7:      learn: 1.9736402        total: 1.99s    remaining: 496ms
8:      learn: 1.9647060        total: 2.27s    remaining: 253ms
9:      learn: 1.9457368        total: 2.54s    remaining: 0us
```

[34]:
```
<catboost.core.CatBoostClassifier at 0x11d39633e08>
```

[45]:
```python
preds = cat.predict(eval_dataset)
```

[48]:
```python
cat_score = accuracy_score(y_test, preds)
```

[48]:
```
0.2862
```

[46]:
```python
print("Accuracy Score: {}".format(accuracy_score(y_test, preds)))
```

```
Accuracy Score: 0.2862
```

[49]:
```python
#Compare accuracy of best XGB Model vs. CatBoost
print('Increase of {:0.2f}%.'.format( 100 * (gb_base_score - cat_score) /␣
↪cat_score))
```

```
Increase of 75.89%.
```

Comparing XGBoost to CatBoost, XGBoost (our best model) was better by an increase of 75.89% in terms of accuracy. This was the best model configured for this problem while CatBoost fell behind.

RFC received a best score of 0.467 while Logistic Regression got around a 0.35. Our best Gradient Boosting got a score of 0.502 which is above the 50% accuracy threshold. Again, the difference in score is probably too small to make a decisive split in future data performance, but we can be slightly more confident in predicting Gradient Boosting to be more accurate.

# 5 Problem 5

**Pytorch Tutorial**

[0]:
```python
from __future__ import print_function
import torch
```

[0]:
```python
x = torch.empty(5, 3)
print(x)
```

```
tensor([[-1.5552e-01,  0.0000e+00,  4.4842e-44],
        [ 0.0000e+00,         nan,  0.0000e+00],
        [ 2.6251e-09,  1.3733e-05,  4.2011e-05],
        [ 4.2491e-05,  3.3429e+21,  5.3934e-05],
        [ 2.1782e-04,  1.6838e+22,  0.0000e+00]])
```

[0]:
```python
x = torch.rand(5, 3)
print(x)
```

```
tensor([[0.1194, 0.3407, 0.5182],
        [0.7104, 0.6167, 0.9554],
        [0.4909, 0.9780, 0.0206],
        [0.0160, 0.3529, 0.6266],
        [0.2853, 0.6049, 0.2395]])
```

[0]:
```python
x = torch.zeros(5, 3, dtype=torch.long)
print(x)
```

```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

[0]:
```python
x = torch.tensor([5.5, 3])
print(x)
```

```
tensor([5.5000, 3.0000])
```

[0]:
```python
x = x.new_ones(5, 3, dtype=torch.double)
print(x)

x = torch.randn_like(x, dtype=torch.float)
print(x)
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
tensor([[-0.2334,  1.1119,  0.0848],
        [ 1.0386, -0.6702,  1.5898],
```

```
        [-1.5128, -0.1279,  1.9291],
        [ 1.5583,  0.4790, -0.3507],
        [-0.4396,  0.8310, -0.1686]])
```

[0]:
```python
print(x.size())
```

```
torch.Size([5, 3])
```

[0]:
```python
y = torch.rand(5, 3)
print(x + y)
```

```
tensor([[-0.2058,  1.3123,  1.0061],
        [ 1.8051, -0.1640,  1.9518],
        [-0.6625,  0.3585,  2.2083],
        [ 2.5186,  1.4590, -0.0456],
        [-0.0410,  1.4877,  0.7460]])
```

[0]:
```python
print(torch.add(x, y))
```

```
tensor([[-0.2058,  1.3123,  1.0061],
        [ 1.8051, -0.1640,  1.9518],
        [-0.6625,  0.3585,  2.2083],
        [ 2.5186,  1.4590, -0.0456],
        [-0.0410,  1.4877,  0.7460]])
```

[0]:
```python
result = torch.empty(5, 3)
torch.add(x, y, out=result)
print(result)
```

```
tensor([[-0.2058,  1.3123,  1.0061],
        [ 1.8051, -0.1640,  1.9518],
        [-0.6625,  0.3585,  2.2083],
        [ 2.5186,  1.4590, -0.0456],
        [-0.0410,  1.4877,  0.7460]])
```

[0]:
```python
y.add_(x)
print(y)
```

```
tensor([[-0.2058,  1.3123,  1.0061],
        [ 1.8051, -0.1640,  1.9518],
        [-0.6625,  0.3585,  2.2083],
        [ 2.5186,  1.4590, -0.0456],
        [-0.0410,  1.4877,  0.7460]])
```

[0]:
```python
print(x[:,1])
```

```
tensor([ 1.1119, -0.6702, -0.1279,  0.4790,  0.8310])
```

[0]:
```python
x = torch.randn(4, 4)
y = x.view(16)
z = x.view(-1, 8)
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

[0]:
```python
x = torch.randn(1)
print(x)
print(x.item())
```

```
tensor([0.7712])
0.7711711525917053
```

[0]:
```python
a = torch.ones(5)
print(a)
```

```
tensor([1., 1., 1., 1., 1.])
```

[0]:
```python
b = a.numpy()
print(b)
```

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

[0]:
```python
a.add_(1)
print(a)
print(b)
```

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

[0]:
```python
import numpy as np
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

```
[2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

[0]:
```python
if torch.cuda.is_available():
    device = torch.device("cuda")
    y = torch.ones_like(x, device=device)
    x = x.to(device)
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))
```

```
tensor([1.7712], device='cuda:0')
tensor([1.7712], dtype=torch.float64)
```

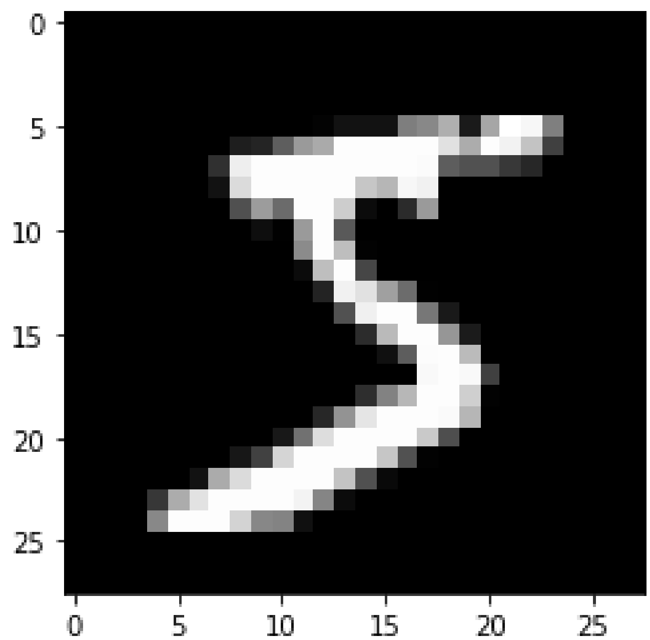
**MNIST Tutorial**

[0]:
```python
from pathlib import Path
import requests

DATA_PATH = Path("data")
PATH = DATA_PATH / "mnist"

PATH.mkdir(parents=True, exist_ok=True)

URL = "http://deeplearning.net/data/mnist/"
FILENAME = "mnist.pkl.gz"

if not(PATH / FILENAME).exists():
  content = requests.get(URL + FILENAME).content
  (PATH / FILENAME).open("wb").write(content)
```

[0]:
```python
import pickle
import gzip

with gzip.open((PATH / FILENAME).as_posix(), "rb") as f:
  ((x_train, y_train), (x_valid, y_valid), _) = pickle.load(f,␣
 ↪encoding="latin-1")
```

[0]:
```python
from matplotlib import pyplot
import numpy as np

pyplot.imshow(x_train[0].reshape((28, 28)), cmap="gray")
print(x_train.shape)
```

```
(50000, 784)
```




```
[0]: import math

     weights = torch.randn(784, 10) / math.sqrt(784)
     weights.requires_grad_()
     bias = torch.zeros(10, requires_grad=True)
```

```
[0]: def log_softmax(x):
       return x - x.exp().sum(-1).log().unsqueeze(-1)

     def model(xb):
       return log_softmax(xb @ weights + bias)
```

```
[0]: bs = 64 # batch size

     xb = x_train[0:bs]  # a mini batch from x
     preds = model(xb)  # predictions
     preds[0], preds.shape
     print(preds[0], preds.shape)
```

```
tensor([-2.2112, -2.7084, -2.5220, -2.3708, -2.3598, -1.8463, -2.6530, -2.1883,
        -2.1134, -2.3597], grad_fn=<SelectBackward>) torch.Size([64, 10])
```

```
[0]: def nll(input, target):
       return -input[range(target.shape[0]), target].mean()

     loss_func = nll
```

[0]:
```python
yb = y_train[0:bs]
print(loss_func(preds, yb))
```

```
tensor(2.3629, grad_fn=<NegBackward>)
```

[0]:
```python
def accuracy(out, yb):
  preds = torch.argmax(out, dim=1)
  return(preds == yb).float().mean()
```

[0]:
```python
print(accuracy(preds, yb))
```

```
tensor(0.0312)
```

[0]:
```python
from IPython.core.debugger import set_trace

lr = 0.5 # learning rate
epochs = 2 # how many epochs to train for

for epoc in range(epochs):
  for i in range((n - 1) // bs + 1):
    # set_trace() # uncomment this to try out debugger
    start_i = i * bs
    end_i = start_i + bs
    xb = x_train[start_i:end_i]
    yb = y_train[start_i:end_i]
    pred = model(xb)
    loss = loss_func(pred, yb)

    loss.backward()
    with torch.no_grad():
      weights -= weights.grad * lr
      bias -= bias.grad * lr
      weights.grad.zero_()
      bias.grad.zero_()
```

[0]:
```python
print(loss_func(model(xb), yb), accuracy(model(xb), yb))
```

```
tensor(0.0809, grad_fn=<NegBackward>) tensor(1.)
```

[0]:
```python
import torch.nn.functional as F

loss_func = F.cross_entropy

def model(xb):
  return xb @ weights + bias
```

[0]:
```python
print(loss_func(model(xb), yb), accuracy(model(xb), yb))
```

```
tensor(0.0809, grad_fn=<NllLossBackward>) tensor(1.)
```

[0]:
```python
from torch import nn

class Mnist_Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.weights = nn.Parameter(torch.randn(784, 10) / math.sqrt(784))
        self.bias = nn.Parameter(torch.zeros(10))

    def forward(self, xb):
        return xb @ self.weights + self.bias
```

[0]:
```python
model = Mnist_Logistic()
```

[0]:
```python
print(loss_func(model(xb), yb))
```

```
tensor(2.4052, grad_fn=<NllLossBackward>)
```

[0]:
```python
with torch.no_grad():
    for p in model.parameters(): p -= p.grad * lr
    model.zero_grad()
```

```
␣
↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most recent call␣
↪last)

      <ipython-input-136-ade642173198> in <module>()
        1 with torch.no_grad():
  ----> 2     for p in model.parameters(): p -= p.grad * lr
        3     model.zero_grad()


      TypeError: unsupported operand type(s) for *: 'NoneType' and 'float'
```

[0]:
```python
def fit():
  for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
      start_i = i * bs
      end_i = start_i + bs
      xb = x_train[start_i: end_i]
      yb = y_train[start_i: end_i]
      pred = model(xb)
```

```
      loss = loss_func(pred, yb)

      loss.backward()
      with torch.no_grad():
        for p in model.parameters():
          p -= p.grad * lr
        model.zero_grad()
fit()
```

[0]:
```
print(loss_func(model(xb), yb))
```

```
tensor(0.0816, grad_fn=<NllLossBackward>)
```

[0]:
```
class Mnist_Logistic(nn.Module):
    def __init__(self):
        super().__init__()
        self.lin = nn.Linear(784, 10)

    def forward(self, xb):
        return self.lin(xb)
```

[0]:
```
model = Mnist_Logistic()
print(loss_func(model(xb), yb))
```

```
tensor(2.3486, grad_fn=<NllLossBackward>)
```

[0]:
```
fit()
print(loss_func(model(xb), yb))
```

```
tensor(0.0808, grad_fn=<NllLossBackward>)
```

[0]:
```
from torch import optim
```

[0]:
```
def get_model():
  model = Mnist_Logistic()
  return model, optim.SGD(model.parameters(), lr=lr)

model, opt = get_model()
print(loss_func(model(xb), yb))

for epoch in range(epochs):
  for i in range((n - 1) // bs + 1):
    start_i = i * bs
    end_i = start_i + bs
    xb = x_train[start_i: end_i]
    yb = y_train[start_i: end_i]
    pred = model(xb)
```

```
    loss = loss_func(pred, yb)

    loss.backward()
    opt.step()
    opt.zero_grad()

print(loss_func(model(xb), yb))
```

```
tensor(2.2887, grad_fn=<NllLossBackward>)
tensor(0.0825, grad_fn=<NllLossBackward>)
```

[0]:
```
from torch.utils.data import TensorDataset
```

[0]:
```
model, opt = get_model()

for epoch in range(epochs):
    for i in range((n - 1) // bs + 1):
        xb, yb = train_ds[i * bs: i * bs + bs]
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        opt.step()
        opt.zero_grad()

print(loss_func(model(xb), yb))
```

```
tensor(0.0818, grad_fn=<NllLossBackward>)
```

[0]:
```
from torch.utils.data import DataLoader

train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs)
```

[0]:
```
for xb, yb in train_dl:
  pred = model(xb)
```

[0]:
```
model, opt = get_model()

for epoch in range(epochs):
  for xb, yb in train_dl:
    pred = model(xb)
    loss = loss_func(pred, yb)

    loss.backward()
    opt.step()
    opt.zero_grad()
```

```python
print(loss_func(model(xb), yb))
```

```
tensor(0.0804, grad_fn=<NllLossBackward>)
```

[0]:
```python
train_ds = TensorDataset(x_train, y_train)
train_dl = DataLoader(train_ds, batch_size=bs, shuffle=True)

valid_ds = TensorDataset(x_valid, y_valid)
valid_dl = DataLoader(valid_ds, batch_size=bs * 2)
```

[0]:
```python
model, opt = get_model()

for epoch in range(epochs):
    model.train()
    for xb, yb in train_dl:
        pred = model(xb)
        loss = loss_func(pred, yb)

        loss.backward()
        opt.step()
        opt.zero_grad()

    model.eval()
    with torch.no_grad():
        valid_loss = sum(loss_func(model(xb), yb) for xb, yb in valid_dl)

    print(epoch, valid_loss / len(valid_dl))
```

```
0 tensor(0.3042)
1 tensor(0.2817)
```

[0]:
```python
def loss_batch(model, loss_func, xb, yb, opt=None):
    loss = loss_func(model(xb), yb)

    if opt is not None:
        loss.backward()
        opt.step()
        opt.zero_grad()

    return loss.item(), len(xb)
```

[0]:
```python
import numpy as np

def fit(epochs, model, loss_func, opt, train_dl, valid_dl):
    for epoch in range(epochs):
        model.train()
```

```
        for xb, yb in train_dl:
            loss_batch(model, loss_func, xb, yb, opt)

        model.eval()
        with torch.no_grad():
            losses, nums = zip(
                *[loss_batch(model, loss_func, xb, yb) for xb, yb in valid_dl]
            )
        val_loss = np.sum(np.multiply(losses, nums)) / np.sum(nums)

        print(epoch, val_loss)
```

[0]:
```
def get_data(train_ds, valid_ds, bs):
    return (
        DataLoader(train_ds, batch_size=bs, shuffle=True),
        DataLoader(valid_ds, batch_size=bs * 2),
    )
```

[53]:
```
train_dl, valid_dl = get_data(train_ds, valid_ds, bs)
model, opt = get_model()
fit(epochs, model, loss_func, opt, train_dl, valid_dl)
```

```
        ␣
↪---------------------------------------------------------------------------

        NameError                                 Traceback (most recent call␣
↪last)

        <ipython-input-53-92ecc9f3d1ee> in <module>()
    ----> 1 train_dl, valid_dl = get_data(train_ds, valid_ds, bs)
          2 model, opt = get_model()
          3 fit(epochs, model, loss_func, opt, train_dl, valid_dl)


        NameError: name 'train_ds' is not defined
```

[0]:
```
class Mnist_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1)
        self.conv2 = nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1)
        self.conv3 = nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1)

    def forward(self, xb):
        xb = xb.view(-1, 1, 28, 28)
```

```
        xb = F.relu(self.conv1(xb))
        xb = F.relu(self.conv2(xb))
        xb = F.relu(self.conv3(xb))
        xb = F.avg_pool2d(xb, 4)
        return xb.view(-1, xb.size(1))

lr = 0.1
```

[0]:
```
model = Mnist_CNN()
opt = optim.SGD(model.parameters(), lr=lr, momentum=0.9)

fit(epochs, model, loss_func, opt, train_dl, valid_dl)
```

```
0 0.3436611232995987
1 0.22426221685409545
```

[0]:
```
class Lambda(nn.Module):
    def __init__(self, func):
        super().__init__()
        self.func = func

    def forward(self, x):
        return self.func(x)


def preprocess(x):
    return x.view(-1, 1, 28, 28)
```

[0]:
```
model = nn.Sequential(
    Lambda(preprocess),
    nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.AvgPool2d(4),
    Lambda(lambda x: x.view(x.size(0), -1)),
)

opt = optim.SGD(model.parameters(), lr=lr, momentum=0.9)

fit(epochs, model, loss_func, opt, train_dl, valid_dl)
```

```
0 0.35687828962802887
1 0.25747167279720307
```

[0]:
```
def preprocess(x, y):
    return x.view(-1, 1, 28, 28), y


class WrappedDataLoader:
    def __init__(self, dl, func):
        self.dl = dl
        self.func = func

    def __len__(self):
        return len(self.dl)

    def __iter__(self):
        batches = iter(self.dl)
        for b in batches:
            yield (self.func(*b))

train_dl, valid_dl = get_data(train_ds, valid_ds, bs)
train_dl = WrappedDataLoader(train_dl, preprocess)
valid_dl = WrappedDataLoader(valid_dl, preprocess)
```

[0]:
```
model = nn.Sequential(
    nn.Conv2d(1, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),
    Lambda(lambda x: x.view(x.size(0), -1)),
)

opt = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
```

[0]:
```
fit(epochs, model, loss_func, opt, train_dl, valid_dl)
```

```
0 0.3401533676624298
1 0.23293600144386292
```

[0]:
```
print(torch.cuda.is_available())
```

```
True
```

[0]:
```
dev = torch.device(
    "cuda") if torch.cuda.is_available() else torch.device("cpu")
```

[0]:
```
def preprocess(x, y):
    return x.view(-1, 1, 28, 28).to(dev), y.to(dev)


train_dl, valid_dl = get_data(train_ds, valid_ds, bs)
train_dl = WrappedDataLoader(train_dl, preprocess)
valid_dl = WrappedDataLoader(valid_dl, preprocess)
```

[0]:
```
model.to(dev)
opt = optim.SGD(model.parameters(), lr=lr, momentum=0.9)
```

[0]:
```
fit(epochs, model, loss_func, opt, train_dl, valid_dl)
```

```
0 0.1917406521320343
1 0.18984941139221193
```

Our Attempt

[64]:
```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

[65]:
```
# hyper parameters
n_epochs = 4
bs_train = 64
bs_test = 1000
lr = 0.3
momentum = 0.5
log_interval = 5

random_seed = 42
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)
```

[65]:
```
<torch._C.Generator at 0x10d9de870>
```

[66]:
```
train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.MNIST(root='./data', train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.1307,), (0.3081,))
                             ])),
  batch_size=bs_train, shuffle=True)

test_loader = torch.utils.data.DataLoader(
```

```
torchvision.datasets.MNIST(root='./data', train=False, download=True,
                           transform=torchvision.transforms.Compose([
                             torchvision.transforms.ToTensor(),
                             torchvision.transforms.Normalize(
                               (0.1307,), (0.3081,))
                           ])),
  batch_size=bs_test, shuffle=True)
```

[67]:
```
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
    self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
    self.conv2_drop = nn.Dropout2d()
    self.lin_layer = nn.Linear(320, 50, bias=False)

  def forward(self, x):
    x = F.relu(F.avg_pool2d(self.conv1(x), 2))
    x = F.relu(F.avg_pool2d(self.conv2_drop(self.conv2(x)), 2))
    x = x.view(-1, 320)
    x = F.relu(self.lin_layer(x))
    x = F.dropout(x, training=self.training)
    return F.log_softmax(x)

MNIST_CNN = CNN()
MNIST_optimizer = optim.SGD(MNIST_CNN.parameters(), lr=lr, momentum=momentum)
```

[68]:
```
def train(epoch):
  MNIST_CNN.train()
  for batch_index, (data, target) in enumerate(train_loader):
    MNIST_optimizer.zero_grad()
    output = MNIST_CNN(data)
    loss = F.nll_loss(output, target)
    loss.backward()
    MNIST_optimizer.step()
```

[69]:
```
def test():
  MNIST_CNN.eval()
  correct = 0
  with torch.no_grad():
    for data, target in test_loader:
      output = MNIST_CNN(data)
      pred = output.data.max(1, keepdim=True)[1]
      correct += pred.eq(target.data.view_as(pred)).sum()
  print('Accuracy: {}%\n'.format(
    100. * correct / len(test_loader.dataset)))
```

[70]:
```
test()
for epoch in range(1, n_epochs + 1):
  train(epoch)
  test()
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:15:
UserWarning: Implicit dimension choice for log_softmax has been deprecated.
Change the call to include dim=X as an argument.
  from ipykernel import kernelapp as app

Accuracy: 0.0%

Accuracy: 92.55000305175781%

Accuracy: 93.4000015258789%

Accuracy: 93.68000030517578%

Accuracy: 93.27999877929688%
```

**Our accuracy is around 93-94%.**

# 6 Problem 6

Note: run this part on GPU

[1]:
```
import torch
import torchvision
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

[2]:
```
# hyper parameters
n_epochs = 4
bs_train = 64
bs_test = 1000
lr = 0.03
momentum = 0.5
log_interval = 10

random_seed = 42
torch.backends.cudnn.enabled = False
torch.manual_seed(random_seed)

device = torch.device(
    "cuda") if torch.cuda.is_available() else torch.device("cpu")
```

[3]:
```
train_loader = torch.utils.data.DataLoader(
  torchvision.datasets.CIFAR10(root='./data', train=True, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                             ])),
  batch_size=bs_train, shuffle=True)

test_loader = torch.utils.data.DataLoader(
  torchvision.datasets.CIFAR10(root='./data', train=False, download=True,
                             transform=torchvision.transforms.Compose([
                               torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize(
                                 (0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                             ])))
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[17]:
```
train_losses = []
test_losses = []
```

[10]:
```
def train_CIFAR10(epoch, model, optimizer):
    model.train()
    loss_total = 0
    for batch_index, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_index % log_interval == 0:
          loss_total += loss.item()
    print(loss_total, len(train_loader.dataset), loss_total / len(train_loader.
 ↪dataset))
    train_losses.append(loss_total / len(train_loader.dataset))
```

[11]:
```
def test_CIFAR10(model):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
```

```
            test_loss += F.nll_loss(output, target, size_average=False).item()
            prediction = output.argmax(dim=1, keepdim=True)
            correct += prediction.eq(target.data.view_as(prediction)).sum()
    test_loss /= len(test_loader.dataset)
    print("test loss:", test_loss)
    test_losses.append(test_loss)
    print('Accuracy: {}%\n'.format(
    100. * correct / len(test_loader.dataset)))
```

Testing different models on CIFAR10

[12]:
```
class Model1(nn.Module):
    def __init__(self):
        super(Model1, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3)
        self.conv2 = nn.Conv2d(64, 128, 3)
        self.conv3 = nn.Conv2d(128, 256, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 4 * 4, 128)
        self.fc2 = nn.Linear(128, 256)
        self.fc3 = nn.Linear(256, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, 64 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)

CNN1 = Model1()
Model1_optimizer = optim.SGD(CNN1.parameters(), lr=lr, momentum=momentum)
CNN1.to(device)
```

[12]:
```
Model1(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (fc1): Linear(in_features=1024, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=10, bias=True)
)
```

```
[13]: # test_CIFAR10(CNN1)
test_losses.clear()
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, CNN1, Model1_optimizer)
  test_CIFAR10(CNN1)
```

```
157.24417734146118 50000 0.0031448835468292236

/opt/anaconda3/lib/python3.7/site-packages/torch/nn/_reduction.py:43:
UserWarning: size_average and reduce args will be deprecated, please use
reduction='sum' instead.
  warnings.warn(warning.format(ret))

test loss: 1.7900823430292307
Accuracy: 36.06999969482422%
```

```
     ␣
↪---------------------------------------------------------------------------

       KeyboardInterrupt                          Traceback (most recent call␣
↪last)

       <ipython-input-13-f3d6b379f980> in <module>
         2 test_losses.clear()
         3 for epoch in range(1, n_epochs + 1):
    ----> 4   train_CIFAR10(epoch, CNN1, Model1_optimizer)
         5   test_CIFAR10(CNN1)


       <ipython-input-10-3322b8f56428> in train_CIFAR10(epoch, model, optimizer)
         7         output = model(data)
         8         loss = F.nll_loss(output, target)
    ----> 9         loss.backward()
        10         optimizer.step()
        11         if batch_index % log_interval == 0:


       /opt/anaconda3/lib/python3.7/site-packages/torch/tensor.py in␣
↪backward(self, gradient, retain_graph, create_graph)
       193                 products. Defaults to ``False``.
       194         """
    --> 195         torch.autograd.backward(self, gradient, retain_graph,␣
↪create_graph)
       196
       197     def register_hook(self, hook):
```

```
/opt/anaconda3/lib/python3.7/site-packages/torch/autograd/__init__.py in␣
↪backward(tensors, grad_tensors, retain_graph, create_graph, grad_variables)
     97     Variable._execution_engine.run_backward(
     98         tensors, grad_tensors, retain_graph, create_graph,
---> 99         allow_unreachable=True)  # allow_unreachable flag
    100
    101


KeyboardInterrupt:
```

Accuracy is around 63% for model 1.

```
[ ]: class Model2(nn.Module):
         def __init__(self):
             super(Model2, self).__init__()
             self.conv1 = nn.Conv2d(3, 12, 3, 1, 1)
             self.relu1 = nn.ReLU()
             self.conv2 = nn.Conv2d(12, 12, 3, 1, 1)
             self.relu2 = nn.ReLU()
             self.pool = nn.MaxPool2d(2)
             self.conv3 = nn.Conv2d(12, 24,3, 1, 1)
             self.relu3 = nn.ReLU()
             self.conv4 = nn.Conv2d(24, 24, 3, 1, 1)
             self.relu4 = nn.ReLU()
             self.fc = nn.Linear(16 * 16 * 24, 10)

         def forward(self, x):
             x = self.conv1(x)
             x = self.relu1(x)
             x = self.conv2(x)
             x = self.relu2(x)
             x = self.pool(x)
             x = self.conv3(x)
             x = self.relu3(x)
             x = self.conv4(x)
             x = self.relu4(x)
             x = x.view(-1, 16 * 16 * 24)
             x = self.fc(x)
             return F.log_softmax(x)
             # return output

     CNN2 = Model2()
     Model2_optimizer = optim.SGD(CNN2.parameters(), lr=.001, momentum=.9) # Changed␣
      ↪lr and momentum here for model 2
     CNN2.to(device)
```

[ ]:
```
# test_CIFAR10(CNN2)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, CNN2, Model2_optimizer)
  test_CIFAR10(CNN2)
```

Accuracy is around 52% for model 2.

[ ]:
```
class Model3(nn.Module):
    def __init__(self):
        super(Model3, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        return F.log_softmax(x)

CNN3 = Model3()
Model3_optimizer = optim.SGD(CNN3.parameters(), lr=lr, momentum=momentum) #␣
 ↪Kept same parameters as model 1
CNN3.to(device)
```

[ ]:
```
# test_CIFAR10(CNN3)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, CNN3, Model3_optimizer)
  test_CIFAR10(CNN3)
```

Accuracy is around 53% for model 3.

The depth of the models does not seem to have a great effect on the result. The models have a varying number of layers, but the accuracy is relatively close. It matters more what the layers are doing, rather than how many.

Testing effect of momentum with learning rate of .1

[ ]:
```
momentum_array = [0.1, 0.3, 0.7, .9]

# these will hold each array for each trial so we can plot later without having␣
 ↪to rerun model
momentum_train_losses1 = []
momentum_test_losses1 = []

momentum_train_losses2 = []
```

```
momentum_test_losses2 = []

momentum_train_losses3 = []
momentum_test_losses3 = []

momentum_train_losses4 = []
momentum_test_losses4 = []

model1 = Model1()
model1.to(device)
train_losses.clear()
test_losses.clear()
model1_optimizer = optim.SGD(model1.parameters(), lr=.1, momentum = 0.1)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model1, model1_optimizer)
  test_CIFAR10(model1)
momentum_train_losses1 = train_losses
momentum_test_losses1 = test_losses

model2 = Model1()
model2.to(device)
train_losses.clear()
test_losses.clear()
model2_optimizer = optim.SGD(model2.parameters(), lr=.1, momentum = 0.3)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model2, model2_optimizer)
  test_CIFAR10(model2)
momentum_train_losses2 = train_losses
momentum_test_losses2 = test_losses


model3 = Model1()
model3.to(device)
train_losses.clear()
test_losses.clear()
model3_optimizer = optim.SGD(model3.parameters(), lr=.1, momentum = 0.7)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model3, model3_optimizer)
  test_CIFAR10(model3)
momentum_train_losses3 = train_losses
momentum_test_losses3 = test_losses


model4 = Model1()
model4.to(device)
train_losses.clear()
test_losses.clear()
```

```
model4_optimizer = optim.SGD(model4.parameters(), lr=.1, momentum = 0.9)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model4, model4_optimizer)
  test_CIFAR10(model4)
momentum_train_losses4 = train_losses
momentum_test_losses4 = test_losses


# NOTE: for some reason these all get overwrriten by last set of losses (maybe␣
↪memory issue with GPU)
print(momentum_train_losses1)
print(momentum_test_losses1)

print(momentum_train_losses2)
print(momentum_test_losses2)

print(momentum_train_losses3)
print(momentum_test_losses3)

print(momentum_train_losses4)
print(momentum_test_losses4)
```

learning rate = .01

[ ]:
```
momentum_array = [0.1, 0.3, 0.7, .9]

# these will hold each array for each trial so we can plot later without having␣
↪to rerun model
momentum_train_losses1 = []
momentum_test_losses1 = []

momentum_train_losses2 = []
momentum_test_losses2 = []

momentum_train_losses3 = []
momentum_test_losses3 = []

momentum_train_losses4 = []
momentum_test_losses4 = []

model1 = Model1()
model1.to(device)
train_losses.clear()
test_losses.clear()
model1_optimizer = optim.SGD(model1.parameters(), lr=.01, momentum = 0.1)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model1, model1_optimizer)
```

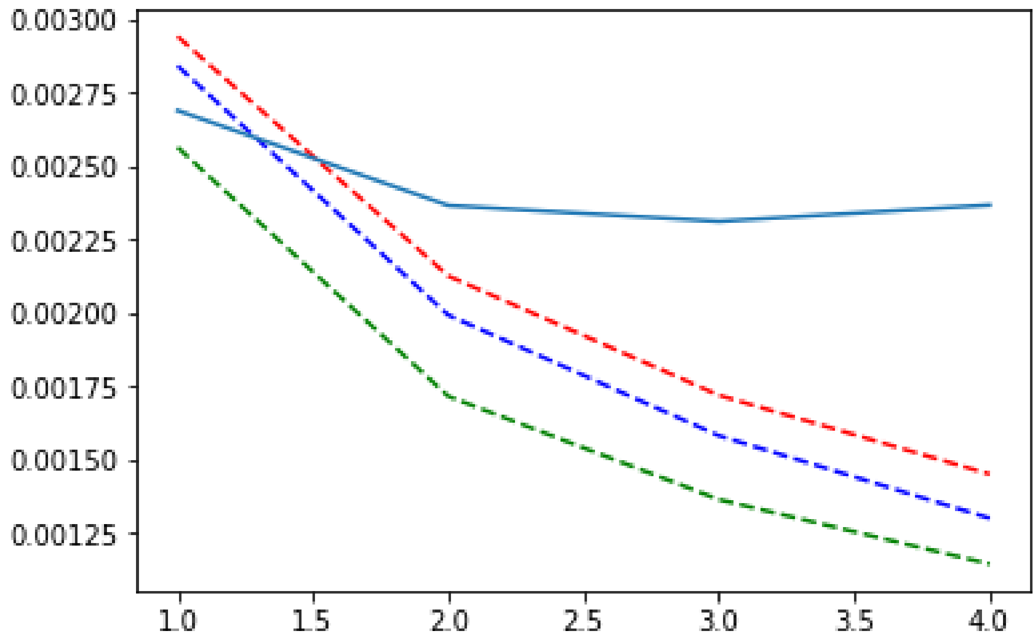
```
  test_CIFAR10(model1)
momentum_train_losses1 = train_losses
momentum_test_losses1 = test_losses

model2 = Model1()
model2.to(device)
train_losses.clear()
test_losses.clear()
model2_optimizer = optim.SGD(model2.parameters(), lr=.01, momentum = 0.3)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model2, model2_optimizer)
  test_CIFAR10(model2)
momentum_train_losses2 = train_losses
momentum_test_losses2 = test_losses


model3 = Model1()
model3.to(device)
train_losses.clear()
test_losses.clear()
model3_optimizer = optim.SGD(model3.parameters(), lr=.01, momentum = 0.7)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model3, model3_optimizer)
  test_CIFAR10(model3)
momentum_train_losses3 = train_losses
momentum_test_losses3 = test_losses


model4 = Model1()
model4.to(device)
train_losses.clear()
test_losses.clear()
model4_optimizer = optim.SGD(model4.parameters(), lr=.01, momentum = 0.9)
for epoch in range(1, n_epochs + 1):
  train_CIFAR10(epoch, model4, model4_optimizer)
  test_CIFAR10(model4)
momentum_train_losses4 = train_losses
momentum_test_losses4 = test_losses
```

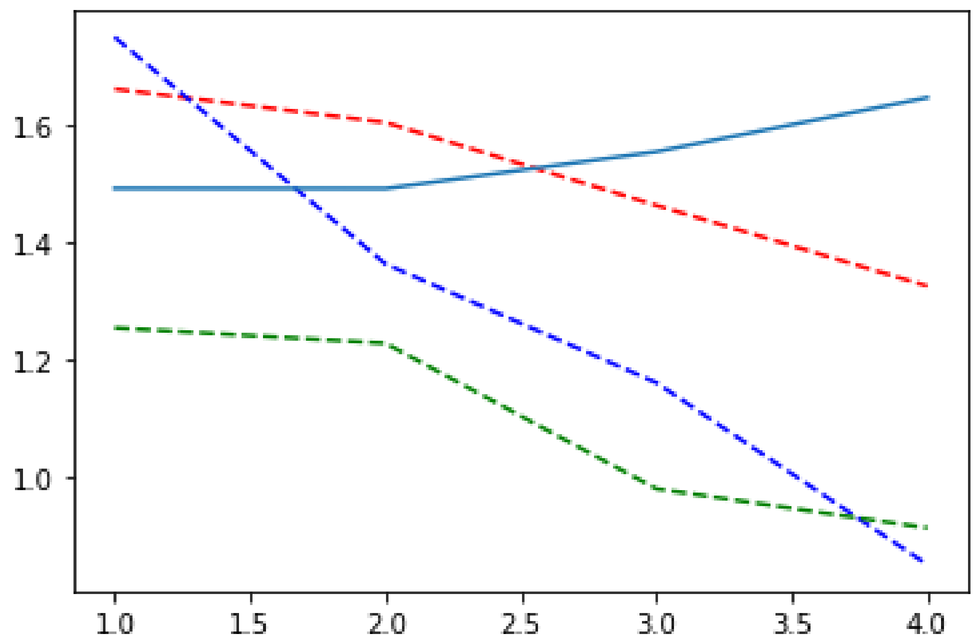
[16]:

```
# Plots for learning rate = 0.1
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
# NOTE: have to copy the losses over after each calculation because list is␣
↪being overwritten for some reason
momentum_train_losses1 = [0.0029403739094734193,0.0021254292941093444,0.
↪001719337958097458, 0.0014503430569171906]
```

```python
momentum_train_losses2 = [0.0028407124972343444, 0.0019919761276245116, 0.
 ↪0015815676379203796, 0.0013008242398500442]
momentum_train_losses3 = [0.0025631234550476073,
0.0017157915306091308,
0.0013644081687927247,
0.0011445720857381821]
momentum_train_losses4 = [0.0026884434604644777,
0.0023653291058540345,
0.0023112612557411195,
0.0023667406845092774]

momentum_test_losses1 = [1.6625812524676322,
1.6051895192146302,
1.4630702455163003,
1.3252057385206222]
momentum_test_losses2 = [1.750878131377697,
1.362992237663269,
1.1594786290407182,
0.848188038122654]
momentum_test_losses3 = [1.2534145884156227,
1.2270465430498123,
0.9780042097449303,
0.9109865257620812]
momentum_test_losses4 = [1.492093264257908,
1.4918513139605523,
1.555112489926815,
1.6477410009384155]


fig = plt.figure()
plt.plot(x, momentum_train_losses1, 'r--', x, momentum_train_losses2, 'b--', x,␣
 ↪momentum_train_losses3, 'g--', x, momentum_train_losses4)

fig2 = plt.figure()
plt.plot(x, momentum_test_losses1, 'r--', x, momentum_test_losses2, 'b--', x,␣
 ↪momentum_test_losses3, 'g--', x, momentum_test_losses4)
```

[16]:
```
[<matplotlib.lines.Line2D at 0x12cdf71d0>,
 <matplotlib.lines.Line2D at 0x12cdf7390>,
 <matplotlib.lines.Line2D at 0x12cdf7590>,
 <matplotlib.lines.Line2D at 0x12cdf77d0>]
```

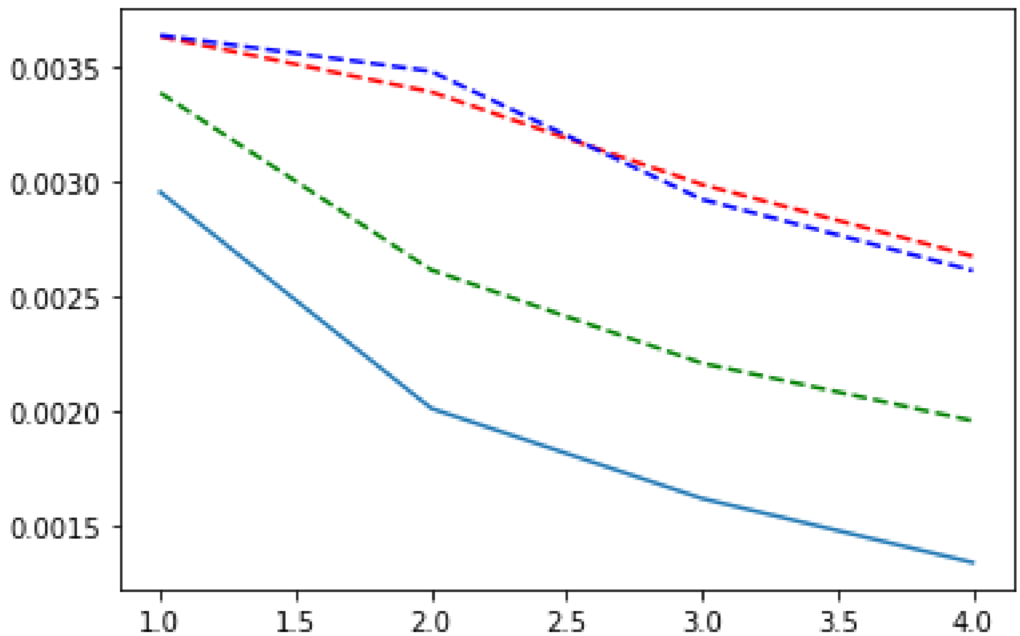


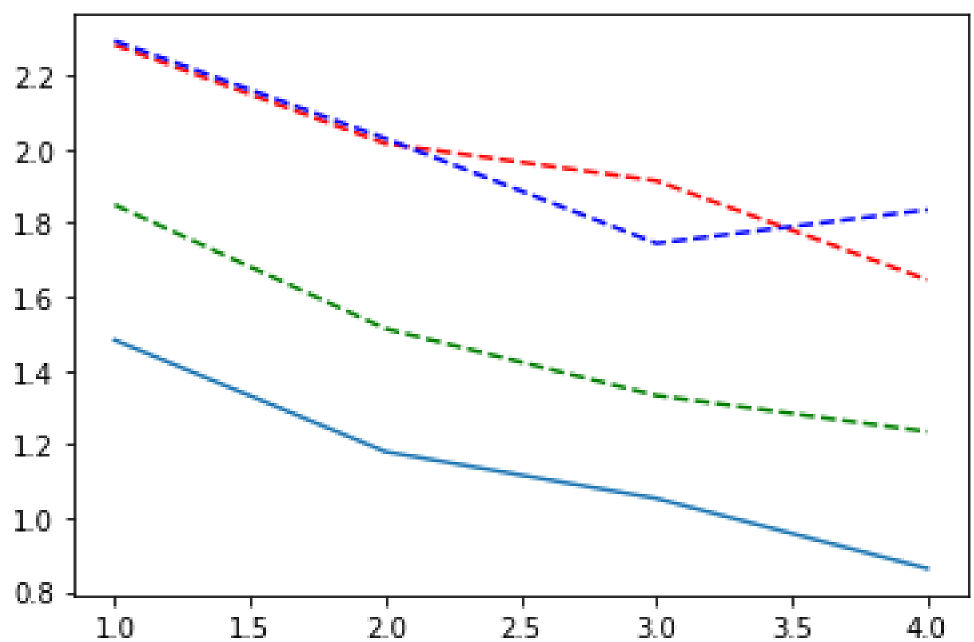




[15]:
```
# Plots for learning rate = 0.01
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
```

```python
# NOTE: have to copy the losses over after each calculation because list is␣
 ↪being overwritten for some reason
momentum_train_losses1 = [0.003628567600250244,
0.003388888680934906,
0.002986208918094635,
0.002674887363910675]
momentum_train_losses2 = [0.0036344224119186403,
0.0034779961729049682,
0.0029218763828277590,
0.0026128575682640076]
momentum_train_losses3 = [0.003384963550567627,
0.002615483522415161,
0.0022098110580444337,
0.001960178356170654]
momentum_train_losses4 = [0.0029532729959487916,
0.0020138071501255034,
0.0016221970522403717,
0.0013435211312770843]

momentum_test_losses1 = [2.2815722467422486,
2.0143521444559096,
1.9135107523322106,
1.645667112469673 1]
momentum_test_losses2 = [2.2925906298160554,
2.028073802471161,
1.7438815403461456,
1.8349284271359443]
momentum_test_losses3 = [1.8480881862998009,
1.512797189950943,
1.331684125316143,
1.2342850183963776]
momentum_test_losses4 = [1.4819285321116447,
1.1797573986291885,
1.0529551633119583,
0.8630687967777252]

# Training Loss
fig = plt.figure()
plt.plot(x, momentum_train_losses1, 'r--', x, momentum_train_losses2, 'b--', x,␣
 ↪momentum_train_losses3, 'g--', x, momentum_train_losses4)

# Testing Loss
fig2 = plt.figure()
plt.plot(x, momentum_test_losses1, 'r--', x, momentum_test_losses2, 'b--', x,␣
 ↪momentum_test_losses3, 'g--', x, momentum_test_losses4)
```

[15]: [<matplotlib.lines.Line2D at 0x12cd6da50>,
 <matplotlib.lines.Line2D at 0x12cd6dc10>,
 <matplotlib.lines.Line2D at 0x12cd6de10>,
 <matplotlib.lines.Line2D at 0x12cd76090>]



From the graphs it is clear that learning rate and momentum have an affect on the losses. With a learning rate of .1, the losses were relatively lower than with a learning rate of .01. In both cases, as momentum increased, for the most part losses decreased.

The best combination of parameters for our model is a learning rate of 0.1 and momentum of 0.3 over 4 epochs. With this combination we got 71% accuracy. If you look at the first set of graphs, the second plot shows this combination having the lowest testing loss as the epochs progress.