CS 4662-01 Advanced Machine Learning Project: Sign Language Recognition Model

Team Members:
Winston Pham: Algorithm Implementation, Image Generation

Norberto Gomez: Algorithm Implementation, Data Analysis, and Research

Nicolas Sandoval: Algorithm Implementation, Python Libraries, and Presentation

Jiajun Gu: Algorithm Implementation, Python Libraries, and Application

Professor: Dr. Mohammad Pourhomayoun

Project Description & Details

- ❖ Our project uses multiple machine learning models to classify Sign Language images and help us identify them as alphabetical letters.
- ❖ The American Sign Language letter database of hand gestures represents a multi-class problem with 24 classes of letters (excluding J and Z which require motion).
- ❖ We used labels with pixel values and compare them with images of signed words to correctly classify the letters
- ❖ The testing/training cases are half the size of a modern MNIST standard but similar to the header row of labels.

Project Goals

- ❖ To create a machine-learning algorithm that can correctly identify hand signs as accurately as possible with the usage of new methods learned from this class.
- ❖ To create a deeper understanding of how to use certain methods and use them to achieve higher accuracy rates by using multiple libraries
- ❖ To improve the Algorithm and increase the accuracy
- ❖ To compare the accuracy of the algorithm with other machine learning algorithms
- ❖ To determine the ROC curve of the multiple algorithms used

Developed Methods, Algorithms, and Tools

- Encoding Method
  - One Hot Encoding

- Algorithms Used
  - CNN
  - Logistic Regression
  - KNN
  - AdaBoost
  - Decision Tree
  - Random Forest

- Tools Used
  - Pandas
  - Numpy
  - Matplotlib
  - Seaborn
  - Sklearn
  - Keras

Developed Codes and Final Results

- Here we grab our dataset from our user files

- Print it so we know what data we are working with

- We show our label and pixel images

```
train_df = pd.read_csv("C:/Users/gnorb/Documents/School/4662/Sign_Language/sign_mnist_train.csv")

train_df.head()
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779
0	3	107	118	127	134	139	143	146	150	153	...	207	207	207	207	206
1	6	155	157	156	156	156	157	156	158	158	...	69	149	128	87	94
2	2	187	188	188	187	187	186	187	188	187	...	202	201	200	199	198
3	2	211	211	212	212	211	210	211	210	210	...	235	234	233	231	230
4	13	164	167	170	172	176	179	180	184	185	...	92	105	105	108	133

5 rows × 785 columns

```
test_df = pd.read_csv("C:/Users/gnorb/Documents/School/4662/Sign_Language/sign_mnist_test.csv")

test_df.head()
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	pixel776	pixel777	pixel778	pixel779
0	6	149	149	150	150	150	151	151	150	151	...	138	148	127	89	82
1	5	126	128	131	132	133	134	135	135	136	...	47	104	194	183	186
2	10	85	88	92	96	105	123	135	143	147	...	68	166	242	227	230
3	0	203	205	207	206	207	209	210	209	210	...	154	248	247	248	253
4	3	188	191	193	195	199	201	202	203	203	...	26	40	64	48	29

5 rows × 785 columns

```
print("Shape of train set: ", train_df.shape, '\n'
      "Shape of test set: ", test_df.shape)
```

```
Shape of train set:  (27455, 785)
Shape of test set:  (7172, 785)
```

Jiajun

Developed Codes and Final Results

- Next we start with CNN (Convolutional Neural Network)
- CNN is useful for machine learning models because they help with image and object recognition
- We grab the letters and plot them on a bar chart A-Z.

Data Visualization

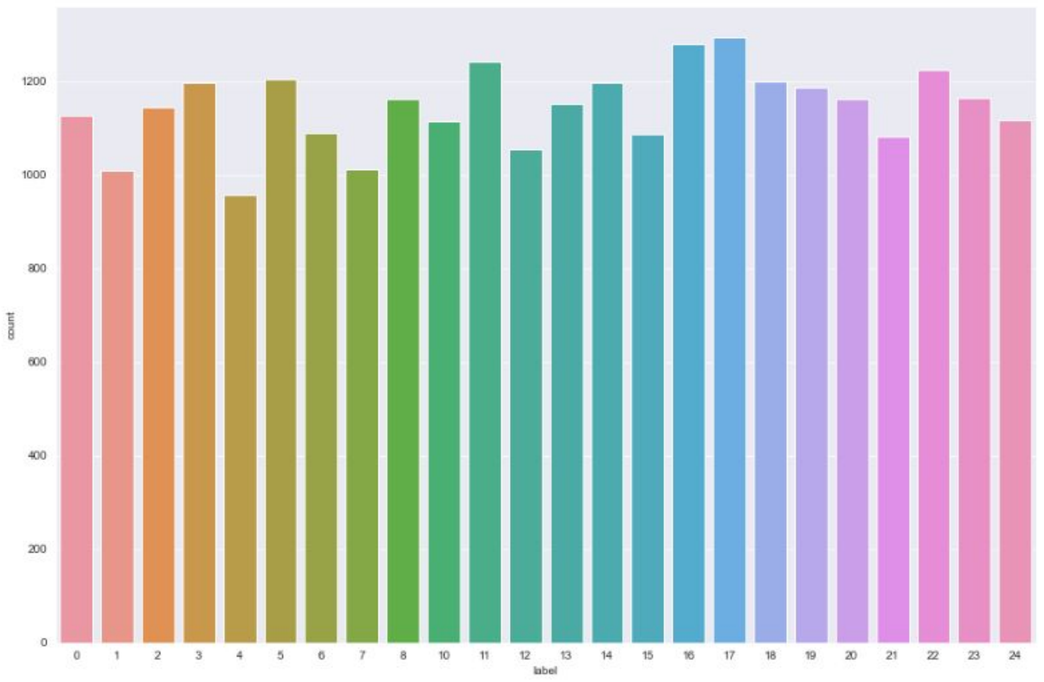
```python
warnings.simplefilter(action='ignore', category=FutureWarning)
labels = train_df['label']
u_labels = np.array(labels)
print(labels)
print(u_labels)
print(np.unique(u_labels))
```

```
0         3
1         6
2         2
3         2
4        13
         ..
27450    13
27451    23
27452    18
27453    17
27454    23
Name: label, Length: 27455, dtype: int64
[ 3  6  2 ... 18 17 23]
[ 0  1  2  3  4  5  6  7  8 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]
```

```python
plt.figure(figsize = (15,10)) # Label Count
sns.set_style("darkgrid")
sns.countplot(train_df['label'])
```

```
<AxesSubplot:xlabel='label', ylabel='count'>
```

Developed Codes and Final Results



Note: The training case represents a label (0-25) as a one-to-one map for each alphabetic letter A-Z (and no cases for 9=J or 25=Z because of gesture motions).

Norberto

Developed Codes and Final Results

- In preprocessing we assigning y_train and y_test with ‘label’ and then delete them off our train_df and test_df.

- This is done in order to have train_df and test_df be composed entirely of just the pixel values.

Preprocessing

```
y_train = train_df['label']
y_test = test_df['label']

del train_df['label']
del test_df['label']

print("Shape of train set: ", train_df.shape, '\n'
      "Shape of test set: ", test_df.shape)
#-----------------------------------------------
print('\n')
print (y_train.shape)
print (y_train[:10])
```

```
Shape of train set:  (27455, 784)
Shape of test set:  (7172, 784)


(27455,)
0     3
1     6
2     2
3     2
4    13
5    16
6     8
7    22
8     3
9     3
Name: label, dtype: int64
```

Developed Codes and Final Results

❖ Once done with preprocessing, we may begin One Hot Encoding on the y_train and y_test labels to categorize the data properly.

```python
# OneHotEncoding for the output label:
y_train = np_utils.to_categorical(y_train, 25)
y_test = np_utils.to_categorical(y_test, 25)

# Label after OneHotEncoding:
print (y_train.shape)
print (y_train[:10,:])
```

```
(27455, 25)
[[0. 0. 0. 1. 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. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0.]
 [0. 0. 1. 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. 1. 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. 1. 0. 0. 0. 0. 0. 0. 0.
  0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 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. 0. 0. 1. 0.
  0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
  0.]]
```

Developed Codes and Final Results

- ❖ After sorting out the labels this is where we start assigning x_train and x_test to solely the pixels from train_df and test_df.

- ❖ We now need to normalize the data and in order to assign it within the range of [0,1], we need to divide the x_train and x_test by 255.

```
x_train = train_df.values
x_test = test_df.values

print(x_train)
print(x_test)
```

```
[[107 118 127 ... 204 203 202]
 [155 157 156 ... 103 135 149]
 [187 188 188 ... 195 194 195]
 ...
 [174 174 174 ... 202 200 200]
 [177 181 184 ...  64  87  93]
 [179 180 180 ... 205 209 215]]
[[149 149 150 ... 112 120 107]
 [126 128 131 ... 184 182 180]
 [ 85  88  92 ... 225 224 222]
 ...
 [190 191 190 ... 211 209 208]
 [201 205 208 ...  67  70  63]
 [173 174 173 ... 195 193 192]]
```

```
# Normalize the data
# simply scale the features to the range of [0,1]:
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train = x_train / 255
x_test = x_test / 255

print(x_train.shape)
print(x_test.shape)
#------------------------
print("\n")
#-------------------------
print(x_train)
print(x_test)
```

```
(27455, 784)
(7172, 784)


[[0.41960785 0.4627451  0.49803922 ... 0.8        0.79607844 0.7921569 ]
 [0.60784316 0.6156863  0.6117647  ... 0.40392157 0.5294118  0.58431375]
 [0.73333335 0.7372549  0.7372549  ... 0.7647059  0.7607843  0.7647059 ]
 ...
 [0.68235296 0.68235296 0.68235296 ... 0.7921569  0.78431374 0.78431374]
 [0.69411767 0.70980394 0.72156864 ... 0.2509804  0.34117648 0.3647059 ]
 [0.7019608  0.7058824  0.7058824  ... 0.8039216  0.81960785 0.84313726]]
[[0.58431375 0.58431375 0.5882353  ... 0.4392157  0.47058824 0.41960785]
 [0.49411765 0.5019608  0.5137255  ... 0.72156864 0.7137255  0.7058824 ]
 [0.33333334 0.34509805 0.36078432 ... 0.88235295 0.8784314  0.87058824]
 ...
 [0.74509805 0.7490196  0.74509805 ... 0.827451   0.81960785 0.8156863 ]
 [0.7882353  0.8039216  0.8156863  ... 0.2627451  0.27450982 0.24705882]
 [0.6784314  0.68235296 0.6784314  ... 0.7647059  0.75686276 0.7529412 ]]
```

Developed Codes and Final Results

- Once we finish the previous steps we may now begin converting the pixels given to us from a 1-D image to a 3-D image.

```
# Reshaping the data from 1-D to 3-D as required through input by CNN's
x_train = x_train.reshape(-1,28,28,1)
x_test = x_test.reshape(-1,28,28,1)

print(x_train.shape)
print(x_test.shape)
#---------------------
print(y_train.shape)
print(y_test.shape)
```

```
(27455, 28, 28, 1)
(7172, 28, 28, 1)
(27455, 25)
(7172, 25)
```

Norberto

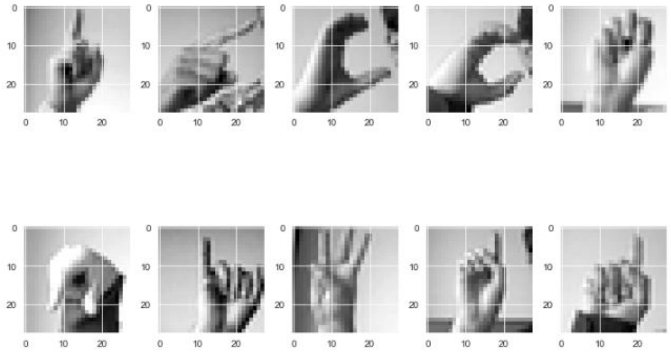
Developed Codes and Final Results

- ❖ Before starting the model training and accuracy tests, we want to first show off a few of the sample images to demonstrate what they will look like and what the model will be trying to interpret.

Preview of first 10 images

Train Images

```
f, ax = plt.subplots(2,5)
f.set_size_inches(10, 10)
k = 0
for i in range(2):
    for j in range(5):
        ax[i,j].imshow(x_train[k].reshape(28, 28) , cmap = "gray")
        k += 1
    plt.tight_layout()
```



Test Images

```
f, ax = plt.subplots(2,5)
f.set_size_inches(10, 10)
k = 0
for i in range(2):
    for j in range(5):
        ax[i,j].imshow(x_test[k].reshape(28, 28) , cmap = "gray")
        k += 1
    plt.tight_layout()
```

Developed Codes and Final Results

Define the Network Architecture (model):

```
# define:
model = Sequential()

# ----------------------------------------
# CNN first layer (with 64 3x3 filter):
model.add(Conv2D(64, kernel_size=(3,3), activation = 'relu', input_shape=(28, 28 ,1), padding="same"))
# with no zero padding -> (None, 26, 26, 64)
# Pooling Layer:
model.add(MaxPooling2D(pool_size = (2, 2)))

# ----------------------------------------
# more hidden layers:
model.add(Conv2D(64, kernel_size = (3, 3), activation = 'relu', padding="same"))
model.add(MaxPooling2D(pool_size = (2, 2)))

# more hidden layers:
model.add(Conv2D(64, kernel_size = (3, 3), activation = 'relu', padding="same"))
# Pooling Layer:
model.add(MaxPooling2D(pool_size = (2, 2)))
# ----------------------------------------
model.add(Flatten())
model.add(Dense(128,activation = "relu"))
# Dropout layer to avoid overfitting
model.add(Dropout(0.20))
model.add(Dense(25, activation = "softmax"))

#compile Model
model.compile(loss="categorical_crossentropy", metrics=["accuracy"], optimizer="adam")
```

```
model.summary()
```

Developed Codes and Final Results

- This is the summary and training for our model
- We begin to train the fitted model to achieve a higher accuracy as the algorithm iterates.

```
Model: "sequential_40"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_120 (Conv2D)         (None, 28, 28, 64)        640

 max_pooling2d_120 (MaxPooli  (None, 14, 14, 64)       0
 ng2D)

 conv2d_121 (Conv2D)         (None, 14, 14, 64)        36928

 max_pooling2d_121 (MaxPooli  (None, 7, 7, 64)         0
 ng2D)

 conv2d_122 (Conv2D)         (None, 7, 7, 64)          36928

 max_pooling2d_122 (MaxPooli  (None, 3, 3, 64)         0
 ng2D)

 flatten_40 (Flatten)        (None, 576)               0

 dense_79 (Dense)            (None, 128)               73856

 dropout_40 (Dropout)        (None, 128)               0

 dense_80 (Dense)            (None, 25)                3225

=================================================================
Total params: 151,577
Trainable params: 151,577
Non-trainable params: 0
_________________________________________________________________
```

```
# 640 = (3x3+1)x64 filters
# 36928 = (3x3x64+1)x64 filters
# 36928 = (3x3x64+1)x64 filters
# 73856 = (576+1)x128
# 3225 = (128+1)x25
```

Training

```
fitted_model = model.fit(x_train, y_train, batch_size= 32, validation_data = (x_test, y_test), epochs= 10, verbose=1)

# batch_size: Integer or None. Number of samples per gradient update.
# epochs: Number of iterations over the entire x and y training data.
# verbose: 0, 1, or 2. how to see the training progress. 0 = silent, 1 = progress bar, 2 = one line per epoch.
# validation_data: Data on which to evaluate the loss and any model metrics at the end of each epoch.
# You can add some callbacks to get a view on internal states and statistics of the model during training:
# https://keras.io/callbacks/
```

```
Epoch 1/10
858/858 [==============================] - 15s 17ms/step - loss: 1.0938 - accuracy: 0.6569 - val_loss: 0.3278 - val_accuracy:
0.8926
Epoch 2/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0907 - accuracy: 0.9709 - val_loss: 0.2597 - val_accuracy:
0.9374
Epoch 3/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0364 - accuracy: 0.9887 - val_loss: 0.2213 - val_accuracy:
0.9455
Epoch 4/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0231 - accuracy: 0.9928 - val_loss: 0.1975 - val_accuracy:
0.9466
Epoch 5/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0264 - accuracy: 0.9912 - val_loss: 0.2269 - val_accuracy:
0.9378
Epoch 6/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0195 - accuracy: 0.9940 - val_loss: 0.2604 - val_accuracy:
0.9414
Epoch 7/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0112 - accuracy: 0.9967 - val_loss: 0.2786 - val_accuracy:
0.9477
Epoch 8/10
858/858 [==============================] - 15s 18ms/step - loss: 0.0181 - accuracy: 0.9941 - val_loss: 0.2116 - val_accuracy:
0.9610
Epoch 9/10
858/858 [==============================] - 15s 18ms/step - loss: 0.0116 - accuracy: 0.9963 - val_loss: 0.2701 - val_accuracy:
0.9350
Epoch 10/10
858/858 [==============================] - 15s 17ms/step - loss: 0.0143 - accuracy: 0.9958 - val_loss: 0.3312 - val_accuracy:
0.9441
```

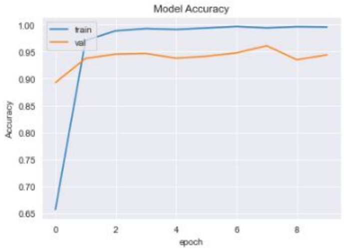
Developed Codes and Final Results

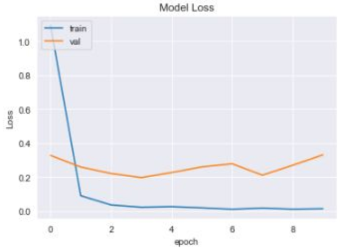
- ❖ After our CNN model, we can see the accuracy of the model in the graph.
- ❖ This also shows us a breakpoint where maybe too much training is happening and we get overfitting.
- ❖ Both model accuracy and model loss show no signs of overfitting with the increase in the number of epochs used.

```python
import matplotlib.pyplot as plt

# summarize history for accuracy
plt.plot(fitted_model.history['accuracy'])
plt.plot(fitted_model.history['val_accuracy'])
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()

# summarize history for loss
plt.plot(fitted_model.history['loss'])
plt.plot(fitted_model.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('epoch')
plt.legend(['train', 'val'], loc='upper left')
plt.show()
```





Norberto

Developed Codes and Final Results

- The overall accuracy with the CNN model we created is 94%.

- This means our model correctly classifies the pixel images into letters at an effective rate of 94%.

Testing

```
# Predictions returned by the model in an array format
predict = model.predict(x_test)

# Returns the indices of the maximum values along an axis.
classes_x = np.argmax(predict, axis=1)

print(classes_x)
```

```
[ 6  5 10 ...  2  4  2]
```

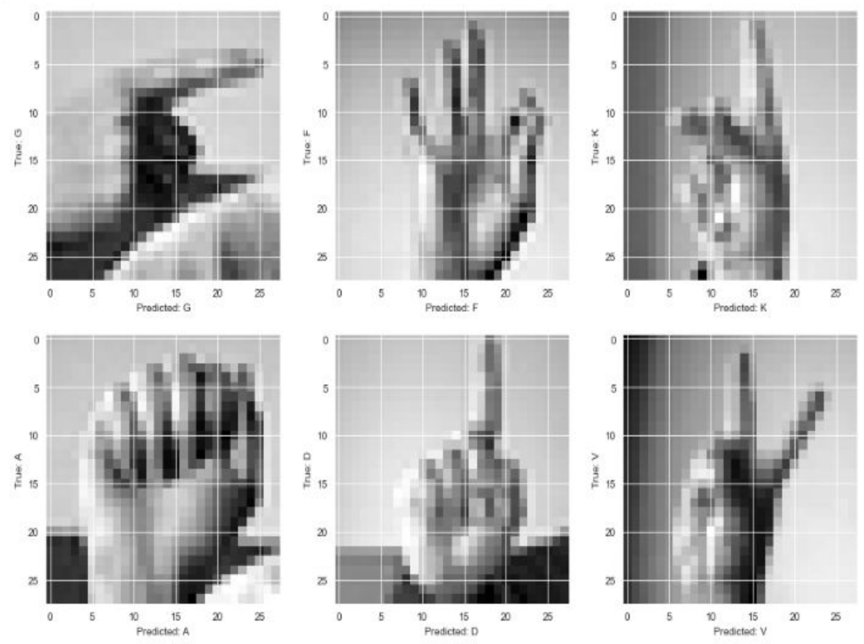
```
score = model.evaluate(x_test, y_test)
print('The accuracy is: ', score[1])
```

```
225/225 [==============================] - 1s 5ms/step - l
The accuracy is:  0.946040153503418
```

```
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

plt.figure(figsize=(15,15))
for i in range(9):
    plt.subplot(3,3,i+1)
    plt.imshow(x_test[i],cmap='gray')
    plt.ylabel(f"True: {alphabet[y[i]]}")
    plt.xlabel(f"Predicted: {alphabet[classes_x[i]]}")

plt.show()
```



Nicolas

Using various Machine Learning algorithms

❖ We read our data and print the shape to visually see the data we are working with.

We need the dataset before removing the "label" column

In [59]:
```
train_df_2 = pd.read_csv("C:/Users/gnorb/Documents/School/4662/Sign_Language/sign_mnist_train.csv")

print(train_df_2.head())
#-----------------------------------------------------------------------
test_df_2  = pd.read_csv("C:/Users/gnorb/Documents/School/4662/Sign_Language/sign_mnist_test.csv")

print(test_df_2.head())
```

```
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0      3     107     118     127     134     139     143     146     150
1      6     155     157     156     156     156     157     156     158
2      2     187     188     188     187     187     186     187     188
3      2     211     211     212     212     211     210     211     210
4     13     164     167     170     172     176     179     180     184

   pixel9  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0     153  ...       207       207       207       207       206       206
1     158  ...        69       149       128        87        94       163
2     187  ...       202       201       200       199       198       199
3     210  ...       235       234       233       231       230       226
4     185  ...        92       105       105       108       133       163

   pixel781  pixel782  pixel783  pixel784
0       206       204       203       202
1       175       103       135       149
2       198       195       194       195
3       225       222       229       163
4       157       163       164       179

[5 rows x 785 columns]
   label  pixel1  pixel2  pixel3  pixel4  pixel5  pixel6  pixel7  pixel8  \
0      6     149     149     150     150     150     151     151     150
1      5     126     128     131     132     133     134     135     135
2     10      85      88      92      96     105     123     135     143
3      0     203     205     207     206     207     209     210     209
4      3     188     191     193     195     199     201     202     203

   pixel9  ...  pixel775  pixel776  pixel777  pixel778  pixel779  pixel780  \
0     151  ...       138       148       127        89        82        96
1     136  ...        47       104       194       183       186       184
2     147  ...        68       166       242       227       230       227
3     210  ...       154       248       247       248       253       236
4     203  ...        26        40        64        48        29        46

   pixel781  pixel782  pixel783  pixel784
0       106       112       120       107
1       184       184       182       180
2       226       225       224       222
3       230       240       253       255
4        49        46        46        53

[5 rows x 785 columns]
```

Preprocessing the data for Classification

```
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score

# Reshape each image pixels into a row of feature table with 28*28=784 features (each pixel is a feature):
x_train = x_train.reshape(27455, 28 * 28 * 1)
x_test = x_test.reshape(7172, 28 * 28 * 1)
# print(x_train)
y_train = train_df_2.iloc[0:27455, 0].values
y_test = test_df_2.iloc[0:7172, 0].values
#----------------------------------------
x_train = x_train * 255
x_test = x_test * 255
#----------------------------------------
print(x_train)
print("\n")
print(y_train)
```

```
[[107. 118. 127. ... 204. 203. 202.]
 [155. 157. 156. ... 103. 135. 149.]
 [187. 188. 188. ... 195. 194. 195.]
 ...
 [174. 174. 174. ... 202. 200. 200.]
 [177. 181. 184. ...  64.  87.  93.]
 [179. 180. 180. ... 205. 209. 215.]]


[ 3  6  2 ... 18 17 23]
```

```
print(x_train.shape)
print(x_test.shape)
#----------------------
print(y_train.shape)
print(y_test.shape)
```

```
(27455, 784)
(7172, 784)
(27455,)
(7172,)
```

Nicolas

Using various Machine Learning algorithms: Logistic Regression

- Using Logistic Regression, we predict an accuracy of 68%

Logistic Regression

In [29]:
```python
from sklearn.linear_model import LogisticRegression

my_logreg = LogisticRegression(max_iter=3000)

# Training ONLY on the training set:
my_logreg.fit(x_train, y_train)

# Testing on the testing set:
y_predict_lr = my_logreg.predict(x_test)

print(y_predict_lr, '\n')

score_lr = accuracy_score(y_test, y_predict_lr)
print('Accuracy:', score_lr, '\n')
```

```
[6 5 9 ... 2 4 2]

Accuracy: 0.6759620747350809
```

In [30]:
```python
y_predict_lr = my_logreg.predict(x_test)
```

```python
y_predict_lr = my_logreg.predict(x_test)
#-----------------------------------------
y_predict_prob_lr = my_logreg.predict_proba(x_test)


# AUC for Logisitc Regression
fpr_lr, tpr_lr, thresholds_lr = metrics.roc_curve(y_test, y_predict_prob_lr[:,1], pos_label=1)

AUC_lr = metrics.auc(fpr_lr, tpr_lr)
print('AUC:', AUC_lr)
```

```
AUC: 0.998963828442686
```

Using various Machine Learning algorithms: KNN

- Using KNN, we predict an accuracy of 80%

KNN

```
from sklearn.neighbors import KNeighborsClassifier

# K value should be odd
k = 3

my_knn = KNeighborsClassifier(n_neighbors=k) # name of the object is arbitrary!

# Training ONLY on the training set:
my_knn.fit(x_train, y_train)

# Testing on the testing set:
y_predict_knn = my_knn.predict(x_test)

print(y_predict_knn, '\n')


score_knn = accuracy_score(y_test, y_predict_knn)
print('Accuracy:', score_knn, '\n')
```

```
[ 6  5 21 ...  2  4  2]

Accuracy: 0.8039598438371445
```

```
y_predict_knn = my_knn.predict(x_test)

y_predict_prob_knn = my_knn.predict_proba(x_test)

# AUC
fpr_knn, tpr_knn, thresholds_knn = metrics.roc_curve(y_test, y_predict_prob_knn[:,1], pos_label=1)

AUC_knn = metrics.auc(fpr_knn, tpr_knn)
print('AUC:', AUC_knn)
```

```
AUC: 0.9785735383009123
```

Using various Machine Learning algorithms: AdaBoost

- Using AdaBoost, we predict an accuracy of 26%
- The lowest accuracy of all our various algorithms

AdaBoost

```
from sklearn.ensemble import AdaBoostClassifier

# In the following line, "my_AdaBoost" is instantiated as an "object" of AdaBoostClassifier "class".

my_AdaBoost = AdaBoostClassifier(n_estimators = 29, random_state=2)

# We can use the method "fit" of the "object my_AdaBoost" along with training dataset and labels to train the model.
my_AdaBoost.fit(x_train, y_train)

y_predict_ab = my_AdaBoost.predict(x_test)

print(y_predict_ab, '\n')

score_ab = accuracy_score(y_test, y_predict_ab)
print('Accuracy:', score_ab, '\n')
```

```
[ 2  5 17 ...  2 12  2]

Accuracy: 0.25697155605131067
```

```
y_predict_ab = my_AdaBoost.predict(x_test)

# This line prints the "estimated likelihood of both label" for the testing set:
y_predict_prob_ab = my_AdaBoost.predict_proba(x_test)
# print(y_predict_prob_ab)

#--------------------------------------------
# AUC for AdaBoost
fpr_ab, tpr_ab, thresholds_ab = metrics.roc_curve(y_test, y_predict_prob_ab[:,1], pos_label= 1)

print(fpr_ab)
print("\n")
print(tpr_ab)
print("\n")
#-------------------------------------------------

AUC_ab = metrics.auc(fpr_ab, tpr_ab)
print('AUC:', AUC_ab)
```

```
[0.00000000e+00 1.48367953e-04 2.96735905e-04 ... 9.99406528e-01
 9.99851632e-01 1.00000000e+00]


[0. 0. 0. ... 1. 1. 1.]


AUC: 0.8650433770194528
```

Using various Machine Learning algorithms: Decision Tree

- Using Decision Tree, we predict an accuracy of 44%

Decision Tree

```
from sklearn.tree import DecisionTreeClassifier

my_DecisionTree = DecisionTreeClassifier(random_state=3)
```

```
# We can use the method "fit" of the objects "my_decisiontree" along with training dataset and labels to train the model.

# Training ONLY on the training set:
my_DecisionTree.fit(x_train, y_train)

# Testing on the testing set:
y_predict_dt = my_DecisionTree.predict(x_test)

print(y_predict_dt, '\n')

# We can now compare the "predicted labels" for the Testing Set with its "actual labels" to evaluate the accuracy
score_dt = accuracy_score(y_test, y_predict_dt)

print('Accuracy:', score_dt, '\n')
```

```
[22 20 10 ...  2  4  2]

Accuracy: 0.43530395984383713
```

```
y_predict_dt = my_DecisionTree.predict(x_test)

y_predict_prob_dt = my_DecisionTree.predict_proba(x_test)

# AUC
fpr_dt, tpr_dt, thresholds_dt = metrics.roc_curve(y_test, y_predict_prob_dt[:,1], pos_label=1)

AUC_dt = metrics.auc(fpr_dt, tpr_dt)
print('AUC:', AUC_dt)
```

```
AUC: 0.758354626882075
```

Using various Machine Learning algorithms: Random Forest

- Using Random Forest, we predict an accuracy of 74%

Random Forest

```
from sklearn.ensemble import RandomForestClassifier

# You can adjust parameters:
my_RandomForest = RandomForestClassifier(n_estimators = 19, bootstrap = True, random_state=3)

# Training ONLY on the training set:
my_RandomForest.fit(x_train, y_train)

# Testing on the testing set:
y_predict_rf = my_RandomForest.predict(x_test)

print(y_predict_dt)
print("\n")
#-------------------------------------------------------
# We can now compare the "predicted labels" for the Testing Set with its "actual labels" to evaluate the accuracy
score_rf = accuracy_score(y_test, y_predict_rf)

print('Accuracy:', score_rf, '\n')
```

```
[22 20 10 ...  2  4  2]


Accuracy: 0.7391243725599553
```

```
y_predict_rf = my_RandomForest.predict(x_test)

y_predict_prob_rf = my_RandomForest.predict_proba(x_test)

# AUC
fpr_rf, tpr_rf, thresholds_rf = metrics.roc_curve(y_test, y_predict_prob_rf[:,1], pos_label=1)

AUC_rf = metrics.auc(fpr_rf, tpr_rf)
print('AUC:', AUC_rf)
```

```
AUC: 0.9878494546103969
```

Graphing the AUC's & ROC curves

ROC

```python
# Importing the "pyplot" package of "matplotlib" library of python to generate
# graphs and plot curves:
import matplotlib.pyplot as plt

# The following line will tell Jupyter Notebook to keep the figures inside the explorer page
# rather than openng a new figure window:
%matplotlib inline

plt.figure()

# Roc Curve:
plt.plot(fpr_lr, tpr_lr, color='red', lw=2,
         label='Logistic Regression (area = %0.2f)' % AUC_lr)
plt.plot(fpr_knn, tpr_knn, color='purple', lw=2,
         label='KNN (area = %0.2f)' % AUC_knn)
plt.plot(fpr_dt, tpr_dt, color='deepskyblue', lw=2,
         label='Decision Tree (area = %0.2f)' % AUC_dt)
plt.plot(fpr_ab, tpr_ab, color='seagreen', lw=2,
         label='AdaBoost (area = %0.2f)' % AUC_ab)
plt.plot(fpr_rf, tpr_rf, color='hotpink', lw=2,
         label='Random Forest (area = %0.2f)' % AUC_rf)

# Random Guess line:
plt.plot([0, 1], [0, 1], color='blue', lw=1, linestyle='--')

# Defining The Range of X-Axis and Y-Axis:
plt.xlim([-0.005, 1.005])
plt.ylim([0.0, 1.01])

# Labels, Title, Legend:
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic')
plt.legend(loc="lower right")

plt.show()
```

- ❖ Logistic Regression has the best performance (1.00)
- ❖ Random Forest (0.99)
- ❖ KNN (0.98)
- ❖ AdaBoost (0.87)
- ❖ Decision Tree (0.76).
- ❖ If we compare these algorithms with our algorithm using CNN, it shows better performance than most classifiers.

Winston

Conclusion

- In order to correctly identify hand signs using CNN, we were able to achieve an extremely high accuracy rate from our algorithm CNN.
- When comparing CNN to other machine learning algorithms we have previously learned and plotting them using the AUC-ROC curve, we can see that CNN is far superior when doing image recognition for this project.
- CNN taught us to use Neural Networks to effectively classifier models that require image classification.