1 Statistical Learning Final Project

1.1 Problem Description

The goal of this model is to distinguish between a picture with a single horse and pictures of non-horses. The goal is to support profile picture uploads to the *Hey Neighbor* horse dating application. *Hey Neighbor* wants to use this model as a pre-screen spam detection mechanism. The intent is to keep other animals and lovers of cars/trucks from submitting profile photos.

1.2 Libraries and Functions

In this section we load libraries needed for the notebook and instatiate functions that will be re-used throughout the notebook.

1.2.1 Libraries

[ ]:
```
import numpy as np
import pandas as pd
import random

import pickle
import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.decomposition import PCA, NMF
from sklearn.preprocessing import StandardScaler, MinMaxScaler
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪roc_curve, roc_auc_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,␣
 ↪QuadraticDiscriminantAnalysis
from sklearn.preprocessing import minmax_scale

# TensorFlow / Keras functions
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import cifar10
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
```

1.2.2 Functions

[ ]:
```
def model_metrics(model_name, true_vals, predicted_vals, predicted_prob_vals =␣
 ↪None):
    """Takes in the model name and calculates metrics we are calculating for␣
 ↪this effort

        Args:
          model_name: The name you want to give the model on the printed output
          true_vals: The true values the model is trying to pedict
          predicted_vals: The predictions the model made
    """

    recall = round(recall_score(true_vals, predicted_vals),2)
    accuracy = round(accuracy_score(true_vals, predicted_vals),2)
    precision = round(precision_score(true_vals, predicted_vals),2)

    print("Recall for the ", model_name, ": ", recall)
    print("Precision for the ", model_name, ": ", precision)
    print("Accuracy for the ", model_name, ": ", accuracy)

    response_data = {"Metric": ['Recall','Precision','Accuracy'],
                     "Values": [recall,precision,accuracy]}

    if predicted_prob_vals is not None:
      fpr, tpr, threshold = roc_curve(true_vals, predicted_prob_vals[:,1])

      plt.plot(fpr, tpr)
      plt.plot([0,1],[0,1])
      plt.ylabel("TPR")
      plt.xlabel("FPR")
```

```
    return pd.DataFrame(response_data)
```

1.3 EDA

In this section we load and explore the images in the CIFAR's data set. The goal is to find relationships in the data that will help us during the modeling phase.

1.3.1 Load and Identify Elements in the Data

Here we load the metadata for the CIFAR-10 data set and the CIFAR-10 data. The metadata provides descriptions about the images we will be analyizing. The `cifar10.load_data()` function provides the CIFAR-10 data into testing and training numpy arrays.

[ ]:
```
# Load up the CIFAR metadata.
with open("cifar_raw_data/batches.meta", 'rb') as fo:
    metadata = pickle.load(fo, encoding='bytes')
labels = metadata[b'label_names']

# Load the CIFAR-10 data into training and testing matrices/arrays
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

classes = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog',␣
 ↪'Horse', 'Ship', 'Truck']
classes_train = np.array([classes[i[0]] for i in y_train])
classes_test = np.array([classes[i[0]] for i in y_test])
```

Lets look at the shapes of the test and training sets.

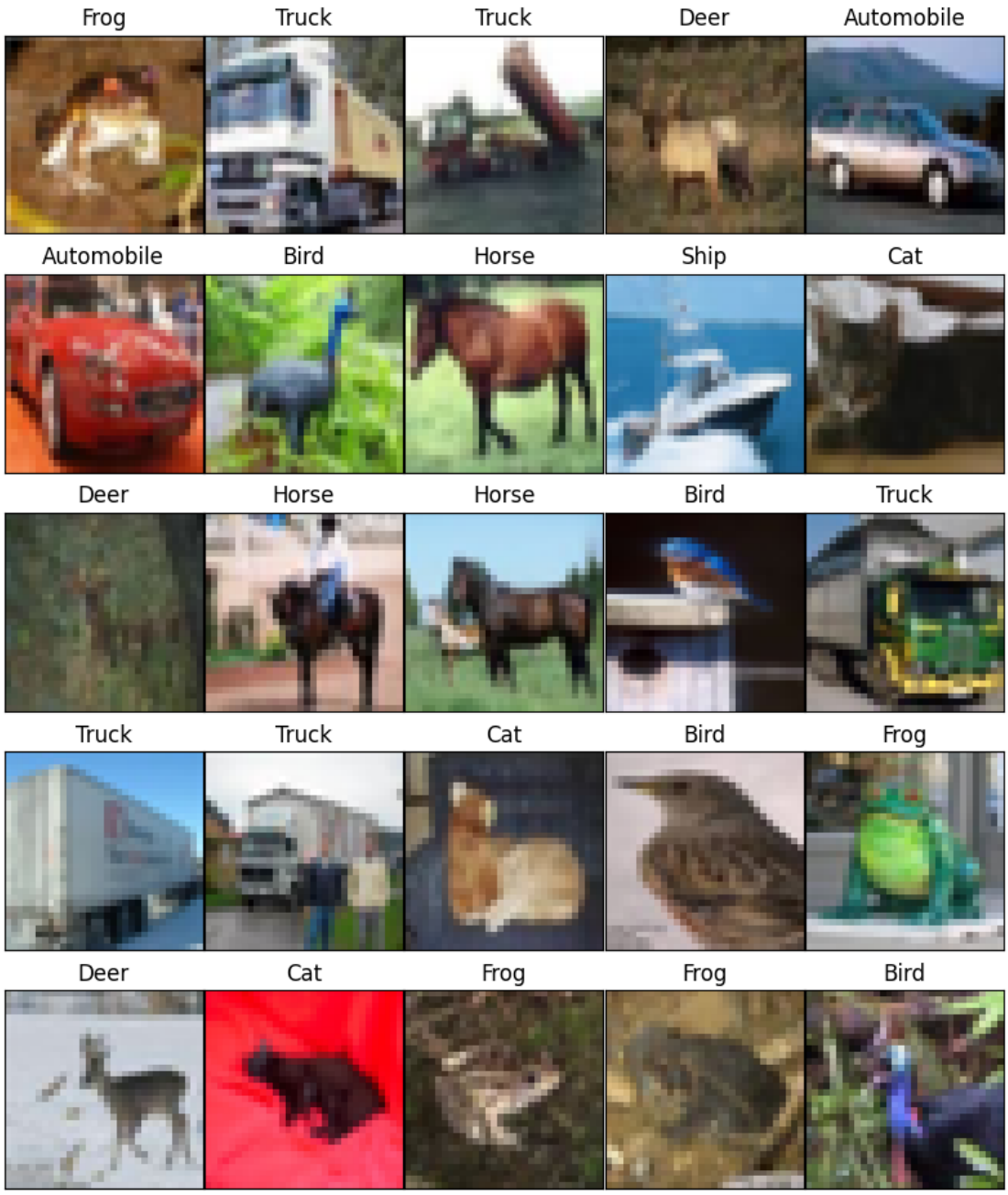
[ ]:
```
print("Dimensions of training features", X_train.shape)
print("Dimensions of test features", X_test.shape)
print("Dimensions of training responses ", y_train.shape)
print("Dimensions of testing responses ", y_test.shape)
```

```
Dimensions of training features (50000, 32, 32, 3)
Dimensions of test features (10000, 32, 32, 3)
Dimensions of training responses  (50000, 1)
Dimensions of testing responses  (10000, 1)
```

Next, lets print a sample of the images

[ ]:
```
fig, axes = plt.subplots(ncols=5, nrows=5, figsize=(10,␣
 ↪12),gridspec_kw=dict(hspace=0.01, wspace=0.01))
index = 0
```

```
for i in range(5):
    for j in range(5):
        axes[i,j].set_title(classes[y_train[index][0]])
        axes[i,j].imshow(X_train[index], cmap='gray')
        axes[i,j].get_xaxis().set_visible(False)
        axes[i,j].get_yaxis().set_visible(False)
        index += 1
plt.show()
```



Fairly low resolution images. Next lets identify what index maps to the image of horses. We can do this by using the metadata for the CIFAR-10 library/package.

The `labels` list contains the names for all the images. The index of the label in the list is number for the response variable that identifies what the image is.

[ ]:
```
labels
```

[ ]:
```
[b'airplane',
 b'automobile',
 b'bird',
 b'cat',
 b'deer',
 b'dog',
 b'frog',
 b'horse',
 b'ship',
 b'truck']
```

We can see that the image we are trying to identify is index 7, horse.

[ ]:
```
labels[7]
```

[ ]:
```
b'horse'
```

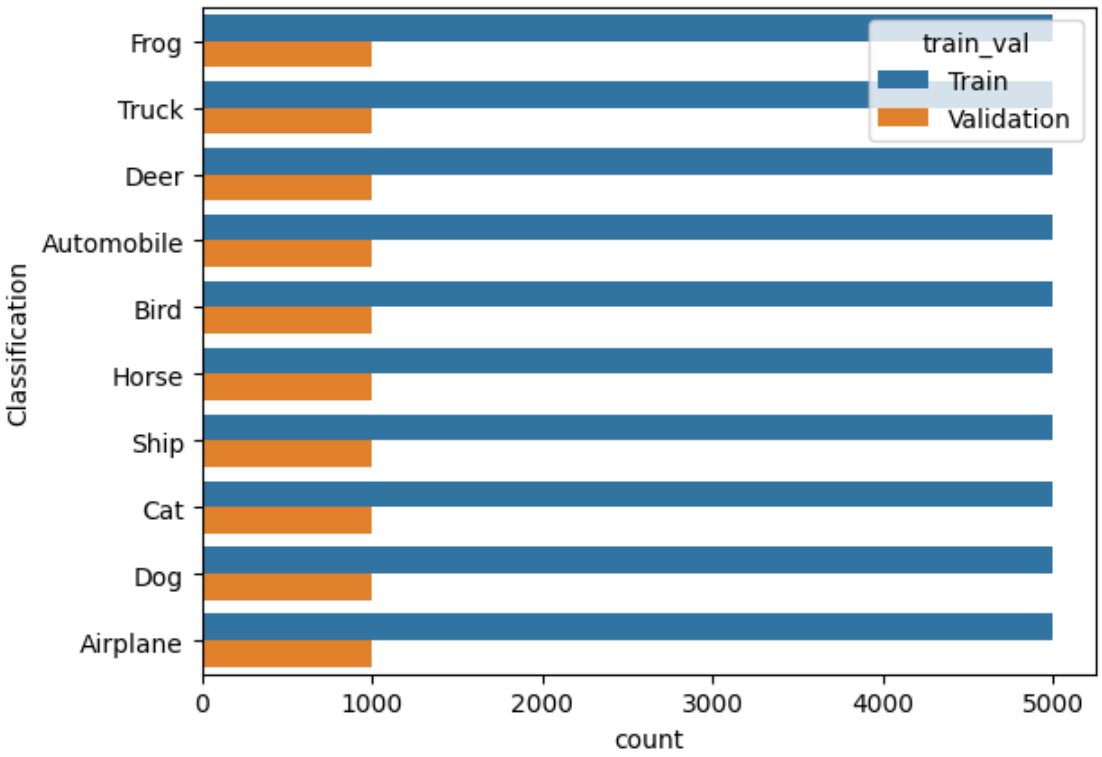
1.3.2 Check the Distribution of the Images in the Datasets

Next we want to look at how many images there are per image type in the total dataset. Want to validate whether or not we have an unbalanced data.

[ ]:
```
#Determine the distribution of data
classes = ['Airplane', 'Automobile', 'Bird', 'Cat', 'Deer', 'Dog', 'Frog',␣
 ↪'Horse', 'Ship', 'Truck']
classes_train = np.array([classes[i[0]] for i in y_train])
classes_test = np.array([classes[i[0]] for i in y_test])

df_train = pd.DataFrame(classes_train,columns = ["Classification"])
df_test = pd.DataFrame(classes_test,columns = ["Classification"])
df_train["train_val"] = "Train"
df_test["train_val"] = "Validation"
df = pd.concat([df_train,df_test])
sns.countplot(y="Classification", data=df, hue='train_val')
```

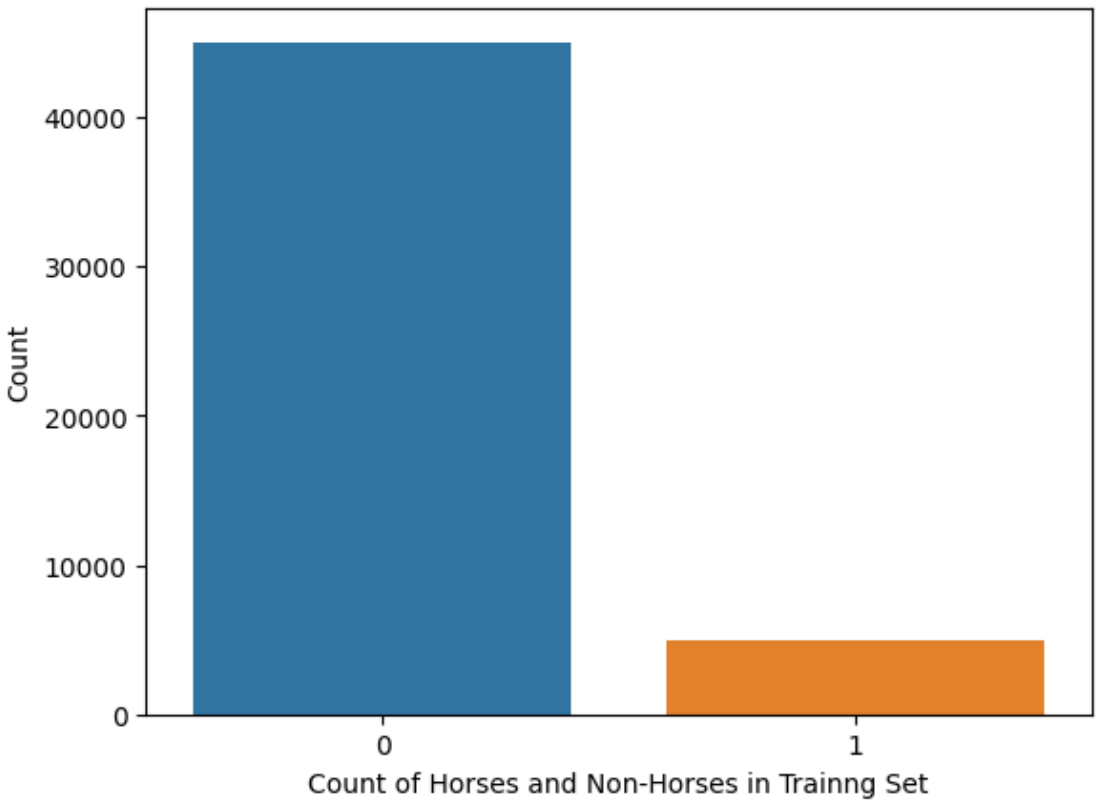
[ ]:
```
<AxesSubplot: xlabel='count', ylabel='Classification'>
```



Next lets look at the comparisons for the number of horses (the class we are trying to predict) to the whole data set.

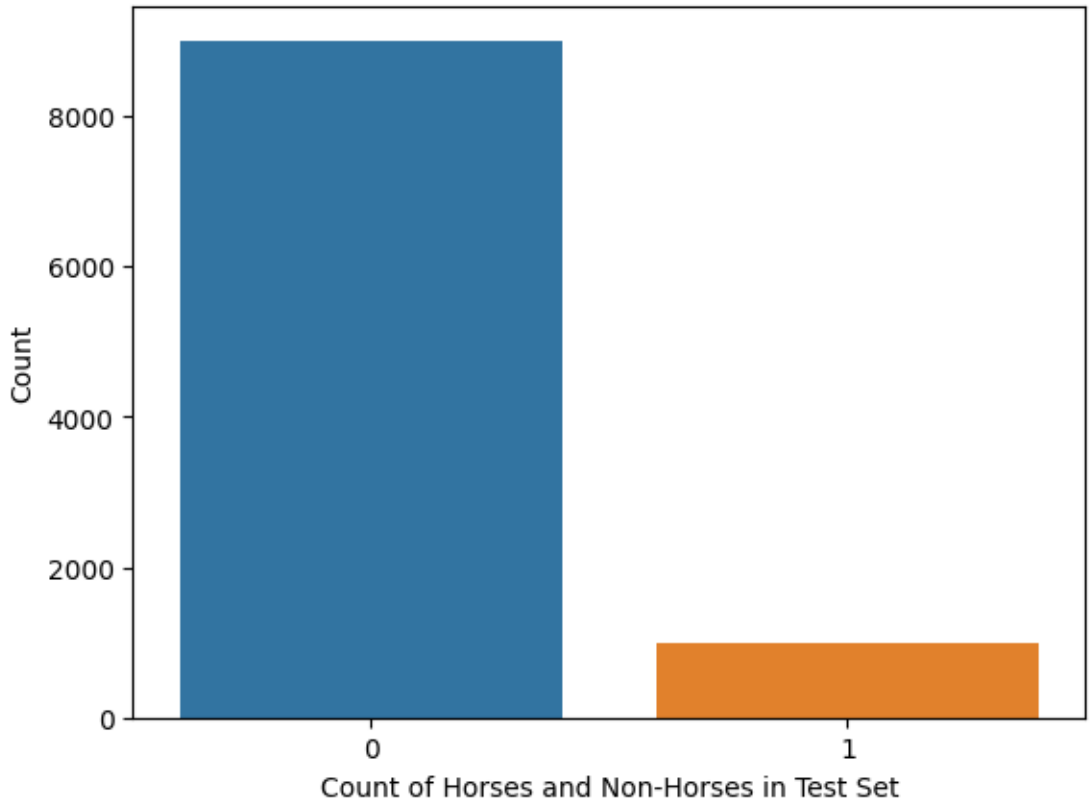
[ ]:
```
df_train['Binary']= np.where(df_train['Classification']== 'Horse',1,0)
df_train['Binary'].value_counts()

ax = sns.countplot(x='Binary', data=df_train)
ax.set(xlabel='Count of Horses and Non-Horses in Trainng Set', ylabel='Count')
plt.show()
```



[ ]:
```
#Count the number of horses in test set
df_test['Binary']= np.where(df_test['Classification']== 'Horse',1,0)
df_test['Binary'].value_counts()

ax = sns.countplot(x='Binary', data=df_test)
ax.set(xlabel='Count of Horses and Non-Horses in Test Set', ylabel='Count')
plt.show()
```



1.3.3 Means and Differences of the images

In this section we compare the means and differences between the different classes of images. The goal is to see if we can identify any interesting elements that will help in the model development process.

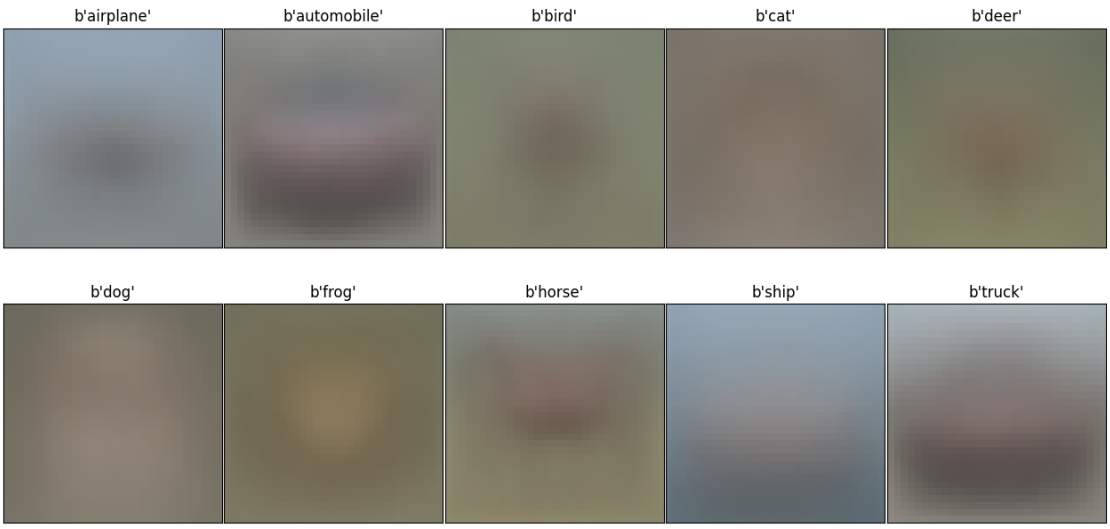
First we'll calculate the means for each class of image. Then will print the image for the means of each class.

[ ]:
```
mean_images = np.empty((10,32,32,3), dtype=int)

for i in range(10):
    image_type_average = np.mean(X_train[np.where(y_train == i)[0]], axis=0,␣
 ↪dtype=int).reshape(32,32,3)
    mean_images[i] = image_type_average
```

[ ]:
```
fig, axes = plt.subplots(ncols=5, nrows=2,␣
 ↪figsize=(16,8),gridspec_kw=dict(hspace=0.01, wspace=0.01))
index = 0
for i in range(2):
    for j in range(5):
        axes[i,j].set_title(labels[index])
```

```
        axes[i,j].imshow(mean_images[index])
        axes[i,j].get_xaxis().set_visible(False)
        axes[i,j].get_yaxis().set_visible(False)
        index += 1
plt.show()
```



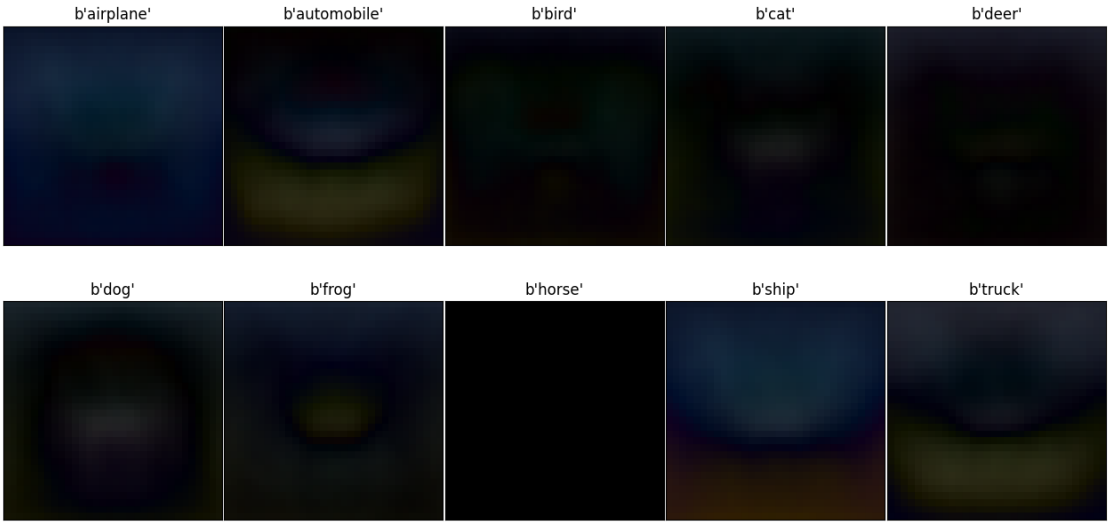
For each class of image you can see the rough outline of the image it represents. For example the car and truck images have roughly the right shape. The horse, roughly looks like a horse.

Next we'll take the difference between our target image (horse) and the other images.

[ ]:
```
diff_images = np.empty((10,32,32,3), dtype=int)

for i in range(10):
    image_diff = mean_images[7] - mean_images[i]
    diff_images[i] = abs(image_diff)

fig, axes = plt.subplots(ncols=5, nrows=2,␣
 ↪figsize=(16,8),gridspec_kw=dict(hspace=0.01, wspace=0.01))
index = 0
for i in range(2):
    for j in range(5):
        axes[i,j].set_title(labels[index])
        axes[i,j].imshow(diff_images[index])
        axes[i,j].get_xaxis().set_visible(False)
        axes[i,j].get_yaxis().set_visible(False)
        index += 1
plt.show()
```



The difference between the mean images, doesn't really provide much insight.

1.3.4 Principal Components and Features

In this section we'll explore if we can use PCA or NMF to reduce the number of features. We'll also look at the images generated by PCA to see if they provide any insight for our modeling.

First we pick a number of components (500) to use as our features. We'll use this value to compute the principal components. Then we'll determine what would be an optimal number of principal components to use, if we were to use PCA.

[ ]:
```
X_train_flatten = X_train.flatten().reshape(50000,3072)
X_test_flatten = X_test.flatten().reshape(10000,3072)

print("Train Flattened Dimensions: ", X_train.shape)
print("Test Flattened Dimensions: ", X_test.shape)
```

```
Train Flattened Dimensions:  (50000, 32, 32, 3)
Test Flattened Dimensions:  (10000, 32, 32, 3)
```

[ ]:
```
pca_cfars = PCA(n_components=500).fit(X_train_flatten)
```

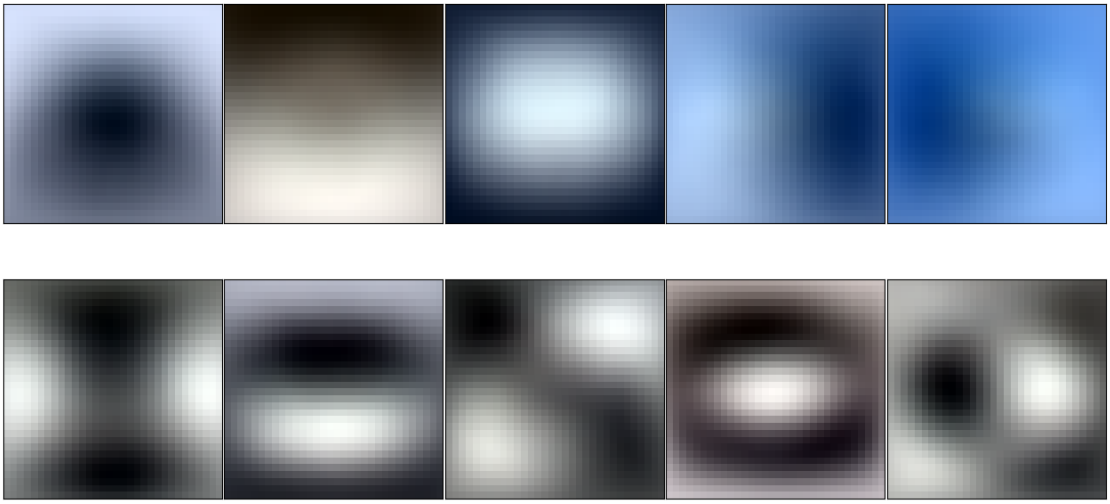
Here we print out the first 10 loadings. Just to get a feel for what the principal components may be.

[ ]:
```
loadings = minmax_scale(pca_cfars.components_, feature_range=(0,1), axis=1)

fig, axes = plt.subplots(ncols=5, nrows=2,␣
 ↪figsize=(16,8),gridspec_kw=dict(hspace=0.01, wspace=0.01))
index = 0
for i in range(2):
```

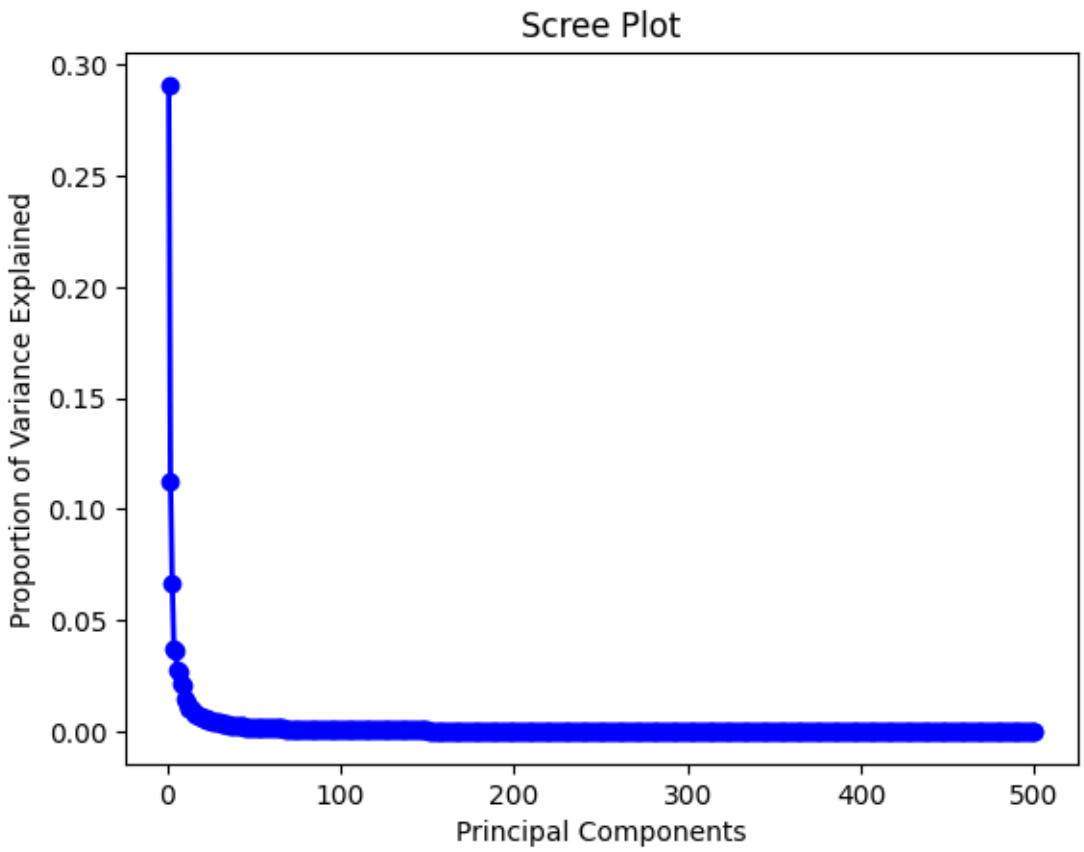
```
    for j in range(5):
        axes[i,j].imshow(loadings[index].reshape(32,32,3))
        axes[i,j].get_xaxis().set_visible(False)
        axes[i,j].get_yaxis().set_visible(False)
        index += 1
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



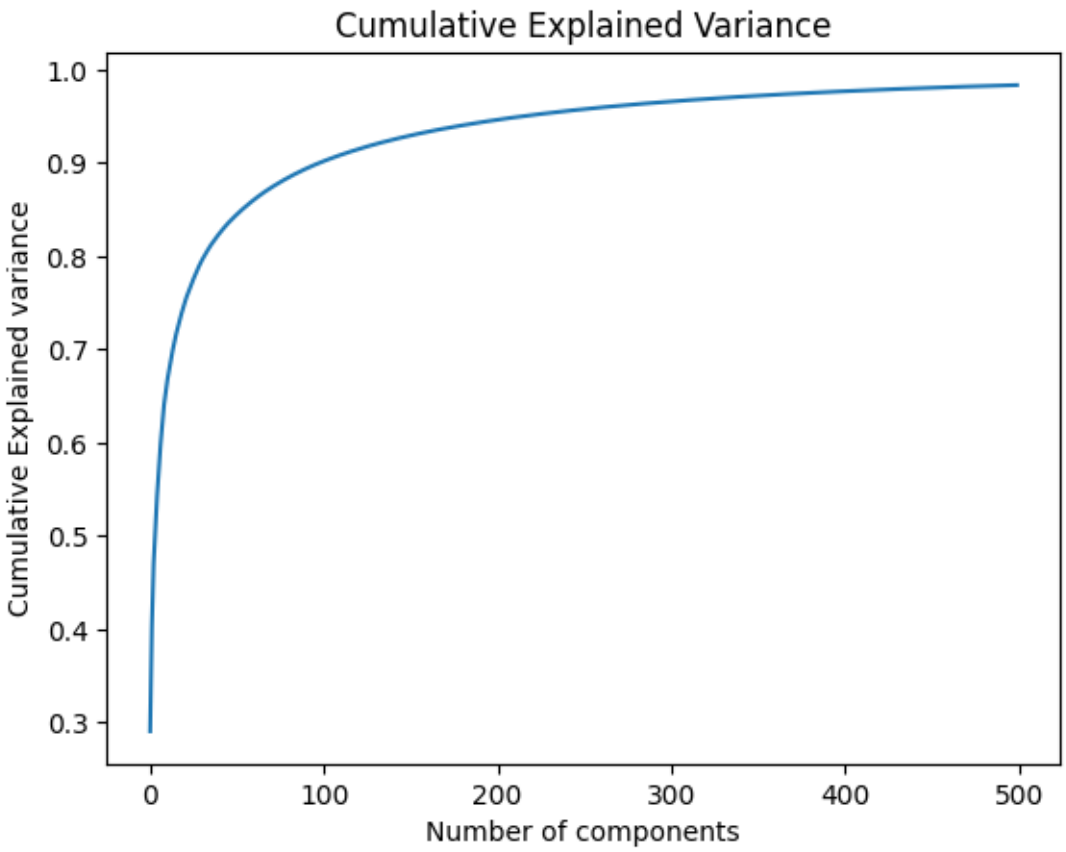
Next we'll pring a scree plot and a cumulative variance plot. The goal is to identify the "optimal" number of principal components we should use. Remember that PCA provides explanations for variance based on features, not necesarily if the features are important.

[ ]:
```
plt.plot(np.arange(pca_cfars.n_components_) + 1, pca_cfars.
 ↪explained_variance_ratio_, 'o-', linewidth=2, color='blue')
plt.title('Scree Plot')
plt.xlabel('Principal Components')
plt.ylabel('Proportion of Variance Explained')
plt.show()
```



[ ]:
```
print('Top 500 components explain {:.0f}% of the variance.'.
  ↪format(100*pca_cfars.explained_variance_ratio_.sum()))
plt.plot(np.cumsum(pca_cfars.explained_variance_ratio_))
plt.title("Cumulative Explained Variance")
plt.xlabel('Number of components')
plt.ylabel('Cumulative Explained variance')
plt.show()
```

```
Top 500 components explain 98% of the variance.
```



500 components we explain 98% of the variability. This is probably suffcient in model selection. For models where this will shortening training time we'll use the PCA created features.

1.3.5 EDA Summary

There are two interesting things found as part of our EDA. First, we can explain 98% of the variablitiy in the images using less than a quarter of the total number of features (2208). This could provide a significant speed improvement should we choose to use PCA for dimensionality reduction.

The second interesting this is the comparison between the means of each class of images. The images from the means for the larger classes (e.g. horse, car, truck, etc.) clearly show shapes similar to the image. Hopefully this will allow our models to more easily identify the classesof images

---

1.4 Initial Models

In this section we'll explore two initial simple models. The goal is to establish a baseline set of results that we can compare more advanced models to.

1.4.1 Random Forest

First we'll use a random forest classifier with the PCA training features to conduct classification. Here we are choosing not do optimizations or hyperparameter tuning. The goal is to use a simple model to establish a baseline.

The Process:

1. First we will create a binary representaiton of the response variable
2. Flatten the training and testing data so that we can use it more effciently with our simple models
3. Scale the training and testing data so that we can use it more effciently with our simple models
4. Use PCA created to create features
5. Train and test the model

[ ]:
```
# Create a binary representation of the data
y_train_binary = np.where(y_train == 7,1,0)
y_test_binary = np.where(y_test == 7,1,0)

print("Number of horses in y_training: ", sum(y_train_binary))
print("Number of horses in y_testing: ", sum(y_test_binary))
```

```
Number of horses in y_training:  [5000]
Number of horses in y_testing:  [1000]
```

[ ]:
```
# Flatten the data
X_train_flatten = X_train.flatten().reshape(50000,3072)
X_test_flatten = X_test.flatten().reshape(10000,3072)

print("Train Flattened Dimensions: ", X_train.shape)
print("Test Flattened Dimensions: ", X_test.shape)
```

```
Train Flattened Dimensions:  (50000, 32, 32, 3)
Test Flattened Dimensions:  (10000, 32, 32, 3)
```

[ ]:
```
# Scal the data using the Min Max Scaler
scale_min_max = MinMaxScaler()
_ = scale_min_max.fit(X_train_flatten)
X_train = scale_min_max.transform(X_train_flatten)
X_test = scale_min_max.transform(X_test_flatten)
```

[ ]:
```
# Create loadings
pca_cfars = PCA(n_components=500).fit(X_train_flatten)
```

[ ]:
```
# Create features from PCA
pca_train_features = pca_cfars.transform(X_train)
pca_test_features = pca_cfars.transform(X_test)
```

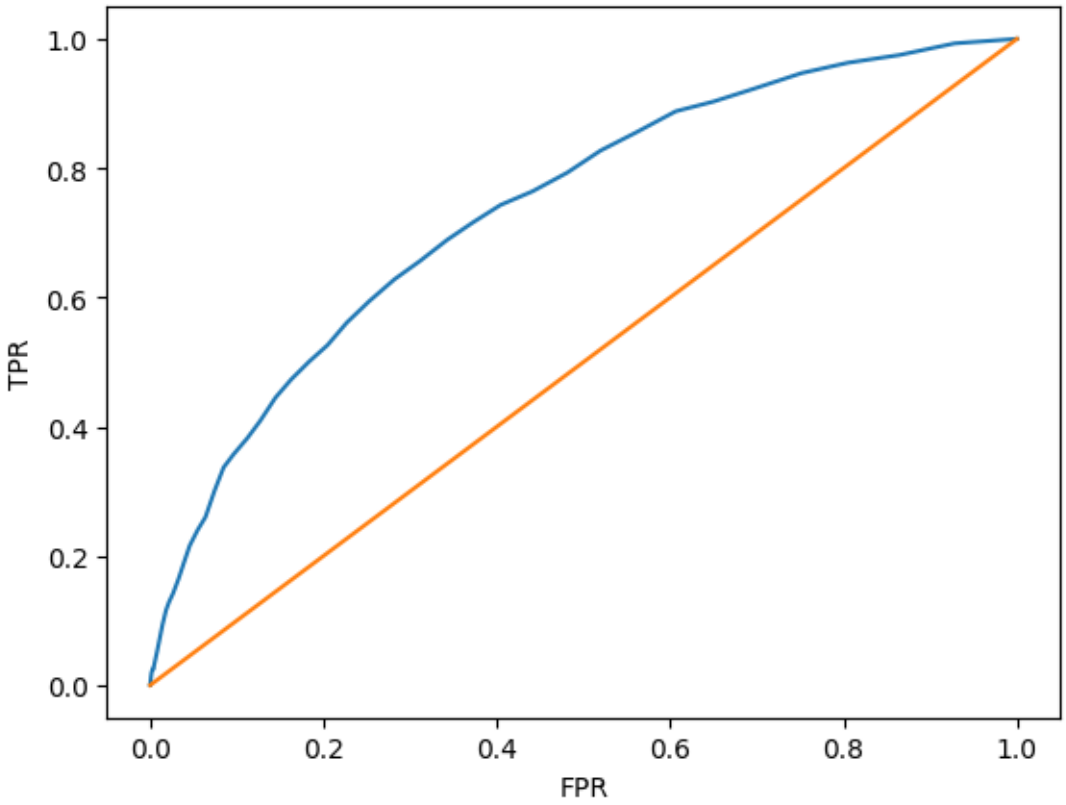
[ ]:
```
# Train Random Forest Classifier
rf = RandomForestClassifier(n_jobs=-1)
_ = rf.fit(pca_train_features,y_train_binary.ravel())
```

[ ]:
```
# Run Predictions
rf_pred_train = rf.predict(pca_train_features)
rf_pred_test = rf.predict(pca_test_features)
rf_preds_prob_train = rf.predict_proba(pca_train_features)
rf_preds_prob_test = rf.predict_proba(pca_test_features)

print("Training Set: --")
model_metrics("Random Forest", y_train_binary,rf_pred_train)
print("\n")
print("Testing Set: --")
_ = model_metrics("Random Forest",␣
 ↪y_test_binary,rf_pred_test,rf_preds_prob_test)
```

```
Training Set: --
Recall for the  Random Forest :  1.0
Precision for the  Random Forest :  1.0
Accuracy for the  Random Forest :  1.0


Testing Set: --
Recall for the  Random Forest :  0.0
Precision for the  Random Forest :  0.62
Accuracy for the  Random Forest :  0.9
```



1.4.2 Quantitative Discriminant Analysis

Here we use a second simple-ish model to create predictions. We'll use the pre-processing done for the random forest classifier (flatten, scale, pca features, etc) for this model.

[ ]:
```
# Train the model
qda = QuadraticDiscriminantAnalysis()
_ = qda.fit(pca_train_features,y_train_binary.ravel())
```

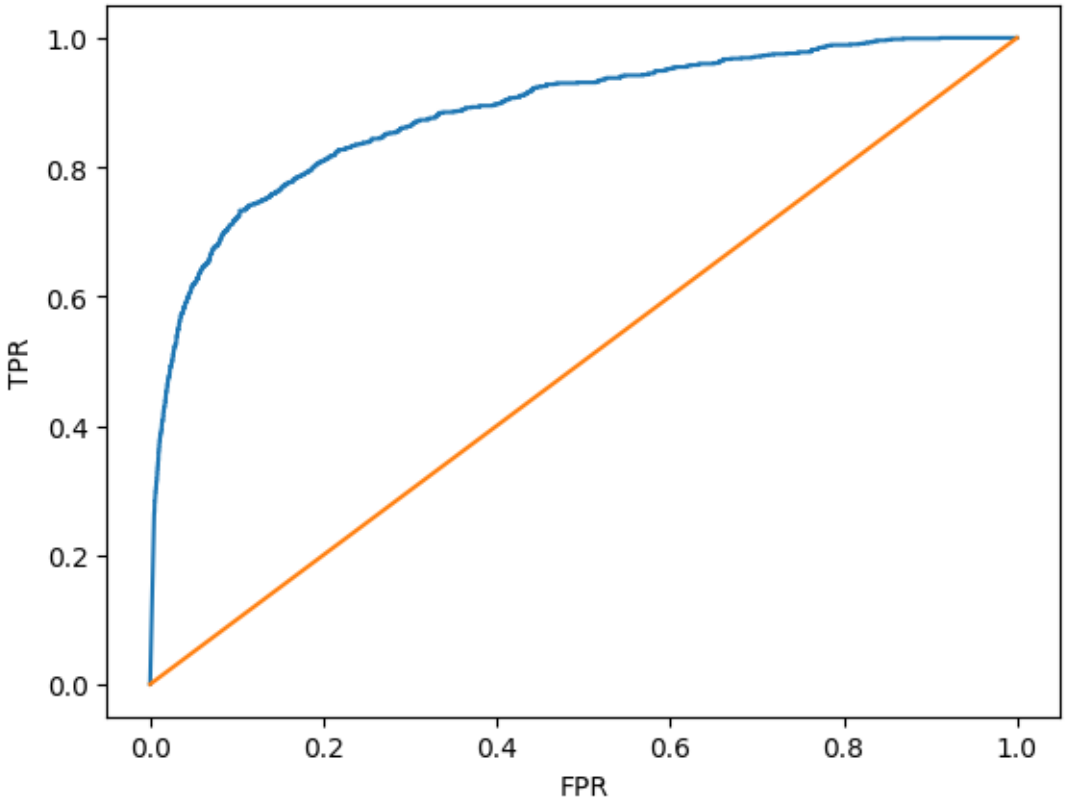
[ ]:
```
# Conduct Predictions
qda_pred_train = qda.predict(pca_train_features)
qda_pred_test = qda.predict(pca_test_features)
qda_pred_prob_test = qda.predict_proba(pca_test_features)

print("Training Set: --")
model_metrics("QDA", y_train_binary,qda_pred_train)
print("\n")
print("Testing Set: --")
_ = model_metrics("QDA", y_test_binary,qda_pred_test,qda_pred_prob_test)
```

```
Training Set: --
Recall for the  QDA :  0.9
```

```
Precision for the  QDA :  0.63
Accuracy for the  QDA :  0.94


Testing Set: --
Recall for the  QDA :  0.66
Precision for the  QDA :  0.52
Accuracy for the  QDA :  0.9
```



1.4.3 Summary Initial Models

Overall the initial models are bad to "just ok". The Random Forrest classifier has 89% accuracy and very low precision and recall. The QDA classifier was better. It had 91% accuracy with 66% recall and 52% precision. While this is better than the Random Forest classifier, it is still not a model I would use for a live system.

[ ]:
```
import numpy as np
import pandas as pd
import random

import pickle
import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.decomposition import PCA, NMF
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪roc_curve, roc_auc_score
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.svm import SVC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis,␣
 ↪QuadraticDiscriminantAnalysis
from sklearn.preprocessing import minmax_scale

# TensorFlow / Keras functions
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.datasets import cifar10
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.layers import Dense, Conv2D, Flatten, MaxPooling2D
from tensorflow.keras.utils import to_categorical
```

1.5 Model Optimization

Here we will use a variant on a sequential Convolutional Neural Network to optimize the results of the model. The model will use a set of 2D convolutional layers and pooling layers to create features from the images.

1.5.1 Pre-Processing

We are minimizing the amount of pre-processing of the data

1. Scale down the test and training data - This is to make the math easier for the model
2. Create binary representation of whether the image is a horse or not a horse - We are trying to detect if the image is a horse or not. 0 if its not a horse, 1 if it is.
3. Create a one hot encoding of the binary representation - This is required for categorization using a CNN

[ ]:
```
# Reload the labels data fresh
with open("cifar_raw_data/batches.meta", 'rb') as fo:
    metadata = pickle.load(fo, encoding='bytes')
```

```
labels = metadata[b'label_names']

# Re-Load the CIFAR-10 data into training and testing matrices/arrays
(X_train, y_train), (X_test, y_test) = cifar10.load_data()

# Scale the training and test data
X_train, X_test = X_train / 255.0, X_test / 255.0
```

[ ]:
```
# Create a binary representation of the data
y_train_binary = np.where(y_train == 7,1,0)
y_test_binary = np.where(y_test == 7,1,0)

y_train_categorical = to_categorical(y_train_binary).astype(int)
y_test_categorical = to_categorical(y_test_binary).astype(int)

print('Example Y variable before transformation:', y_train_binary[0])
print('Example Y variable after transformation:', y_train_categorical[2])
```

```
Example Y variable before transformation: [0]
Example Y variable after transformation: [1 0]
```

1.5.2 Model Creation

[ ]:
```
model1 = Sequential()

model1.add(Conv2D(16, kernel_size=2, activation='relu',padding='same',␣
 ↪input_shape=(32,32,3)))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=2))

model1.add(Conv2D(32, kernel_size=2, activation='relu',padding='same',␣
 ↪input_shape=(32,32,3)))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=2))

model1.add(Conv2D(64, kernel_size=3, activation='relu',padding='same',␣
 ↪input_shape=(32,32,3)))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=2))

model1.add(Conv2D(128, kernel_size=3, activation='relu',padding='same',␣
 ↪input_shape=(32,32,3)))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=2))

model1.add(Conv2D(256, kernel_size=2, activation='relu',padding='same',␣
 ↪input_shape=(32,32,3)))
model1.add(MaxPooling2D(pool_size=(2, 2), strides=2))

model1.add(Flatten())
model1.add(Dense(50, activation='relu'))
```

```
model1.add(Dense(100, activation='relu'))
model1.add(Dense(200, activation='relu'))
model1.add(Dense(300, activation='relu'))

model1.add(Dense(2, activation='softmax'))

model1.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 16)        208

 max_pooling2d (MaxPooling2D  (None, 16, 16, 16)       0
 )

 conv2d_1 (Conv2D)           (None, 16, 16, 32)        2080

 max_pooling2d_1 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_2 (Conv2D)           (None, 8, 8, 64)          18496

 max_pooling2d_2 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 conv2d_3 (Conv2D)           (None, 4, 4, 128)         73856

 max_pooling2d_3 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 conv2d_4 (Conv2D)           (None, 2, 2, 256)         131328

 max_pooling2d_4 (MaxPooling  (None, 1, 1, 256)        0
 2D)

 flatten (Flatten)           (None, 256)               0

 dense (Dense)               (None, 50)                12850

 dense_1 (Dense)             (None, 100)               5100

 dense_2 (Dense)             (None, 200)               20200

 dense_3 (Dense)             (None, 300)               60300
```

```
 dense_4 (Dense)             (None, 2)                 602

=================================================================
Total params: 325,020
Trainable params: 325,020
Non-trainable params: 0
_________________________________________________________________
2022-10-30 14:48:05.329640: W
tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load
dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object
file: No such file or directory
2022-10-30 14:48:05.329666: W
tensorflow/stream_executor/cuda/cuda_driver.cc:263] failed call to cuInit:
UNKNOWN ERROR (303)
2022-10-30 14:48:05.329683: I
tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not
appear to be running on this host (battlewagon): /proc/driver/nvidia/version
does not exist
2022-10-30 14:48:05.329946: I tensorflow/core/platform/cpu_feature_guard.cc:193]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F AVX512_VNNI FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

1.5.3 Model Compilation

- Optimizer - Adam
  - is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments
- Loss - categorical crossentropy
  - Computes the crossentropy loss between the labels and predictions. We should Use this crossentropy loss function when there are two or more label classes.
- Metric - accuracy
  - Best overal metric for this use case

[ ]:
```
model1.compile(optimizer='adam',
               loss='categorical_crossentropy',
               metrics=['accuracy'])
```

[ ]:
```
history = model1.fit(X_train, y_train_categorical, epochs=10, batch_size=500,
                     validation_data=(X_test, y_test_categorical))
```

```
Epoch 1/10
100/100 [==============================] - 8s 72ms/step - loss: 0.3152 -
accuracy: 0.8951 - val_loss: 0.2377 - val_accuracy: 0.9168
Epoch 2/10
100/100 [==============================] - 7s 70ms/step - loss: 0.2212 -
```

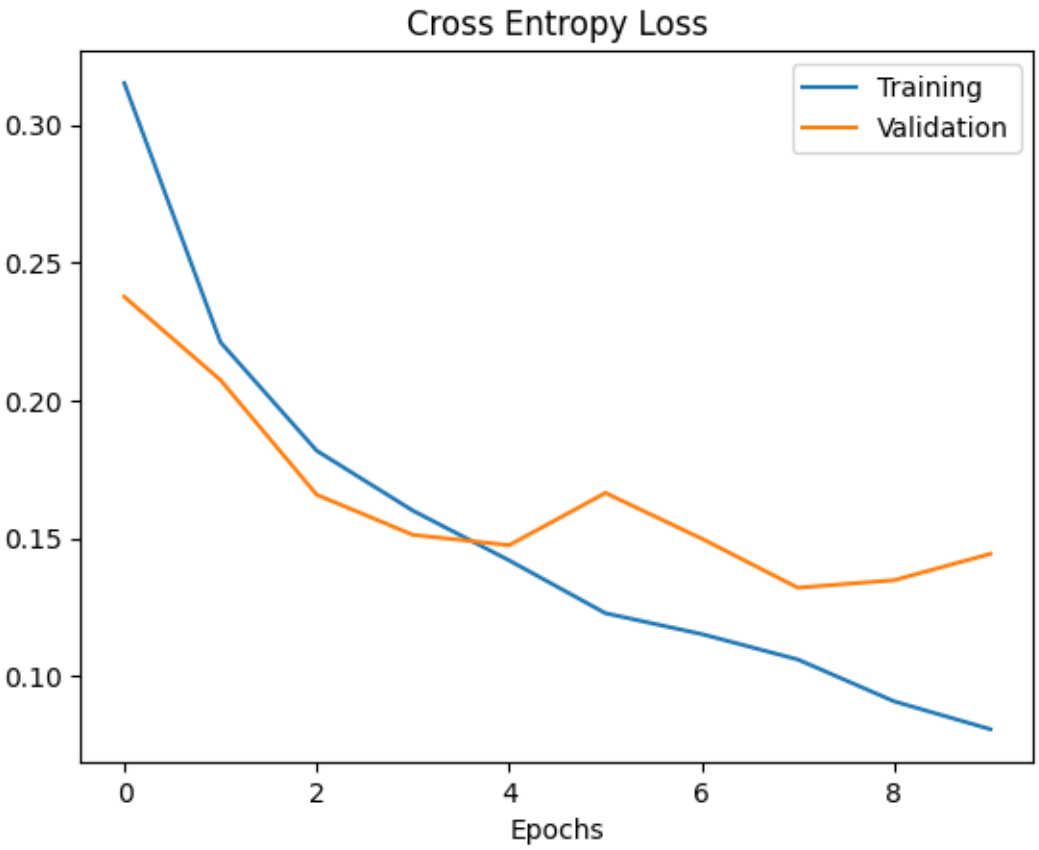
```
accuracy: 0.9200 - val_loss: 0.2074 - val_accuracy: 0.9283
Epoch 3/10
100/100 [==============================] - 7s 68ms/step - loss: 0.1818 -
accuracy: 0.9346 - val_loss: 0.1658 - val_accuracy: 0.9409
Epoch 4/10
100/100 [==============================] - 7s 73ms/step - loss: 0.1600 -
accuracy: 0.9419 - val_loss: 0.1512 - val_accuracy: 0.9455
Epoch 5/10
100/100 [==============================] - 7s 69ms/step - loss: 0.1419 -
accuracy: 0.9490 - val_loss: 0.1475 - val_accuracy: 0.9457
Epoch 6/10
100/100 [==============================] - 7s 74ms/step - loss: 0.1228 -
accuracy: 0.9559 - val_loss: 0.1664 - val_accuracy: 0.9390
Epoch 7/10
100/100 [==============================] - 6s 63ms/step - loss: 0.1152 -
accuracy: 0.9585 - val_loss: 0.1499 - val_accuracy: 0.9498
Epoch 8/10
100/100 [==============================] - 7s 75ms/step - loss: 0.1060 -
accuracy: 0.9619 - val_loss: 0.1320 - val_accuracy: 0.9544
Epoch 9/10
100/100 [==============================] - 7s 68ms/step - loss: 0.0908 -
accuracy: 0.9677 - val_loss: 0.1348 - val_accuracy: 0.9518
Epoch 10/10
100/100 [==============================] - 7s 72ms/step - loss: 0.0806 -
accuracy: 0.9713 - val_loss: 0.1443 - val_accuracy: 0.9533
```

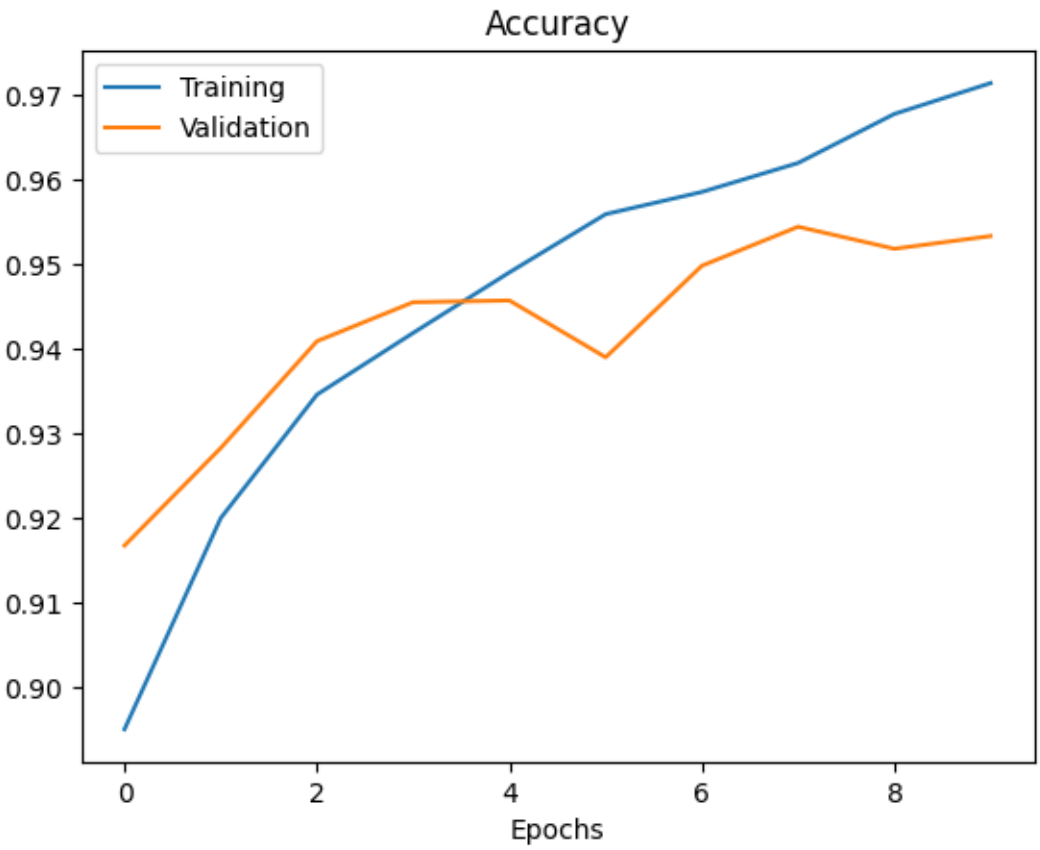
1.5.4 Cross Entropy Loss and Accuracy

Overall it looks like we only need 5ish epochs for training.

[ ]:
```
plt.title('Cross Entropy Loss')
plt.plot(history.history['loss'], label='Training')
plt.plot(history.history['val_loss'], label='Validation')
plt.xlabel('Epochs')
plt.legend()
plt.show()

plt.title('Accuracy')
plt.plot(history.history['accuracy'], label='Training')
plt.plot(history.history['val_accuracy'], label='Validation')
plt.xlabel('Epochs')
plt.legend()
plt.show()
```


Cross Entropy Loss
Training
Validation
0.30
0.25
0.20
0.15
0.10
0
2
4
6
8
Epochs

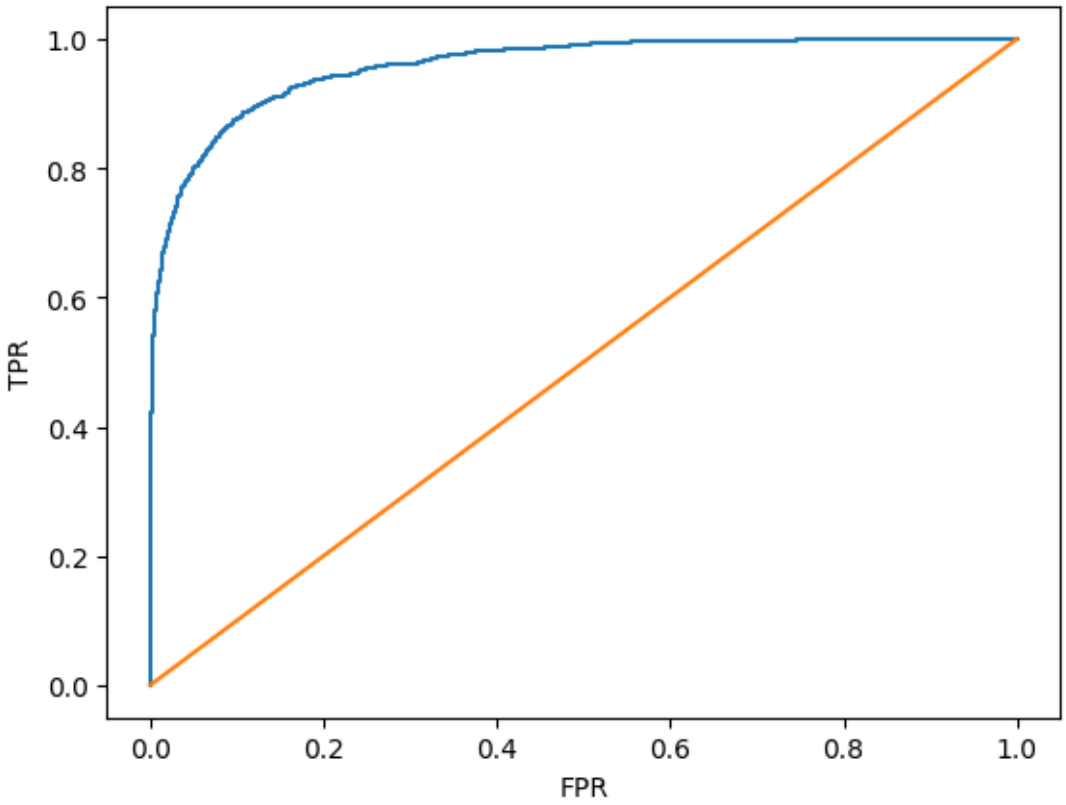


[ ]:
```
test_preds_probability = model1.predict(X_test)
tmp = model_metrics("Sequential CNN", y_test_categorical[:,1]␣
 ↪,test_preds_probability[:,1].round(),test_preds_probability)
tmp
```

```
313/313 [==============================] - 1s 3ms/step
Recall for the  Sequential CNN :  0.65
Precision for the  Sequential CNN :  0.85
Accuracy for the  Sequential CNN :  0.95
```

[ ]:

	Metric	Values
0	Recall	0.65
1	Precision	0.85
2	Accuracy	0.95



1.5.5 Model Optimization Summary

Overall this model is significantly better the base models. Accuracy, precision and recall are significantly better.