Week 5 Progress Report

Presenter: Matt Behnke
Mentor: Dr. Guo

Table of Contents

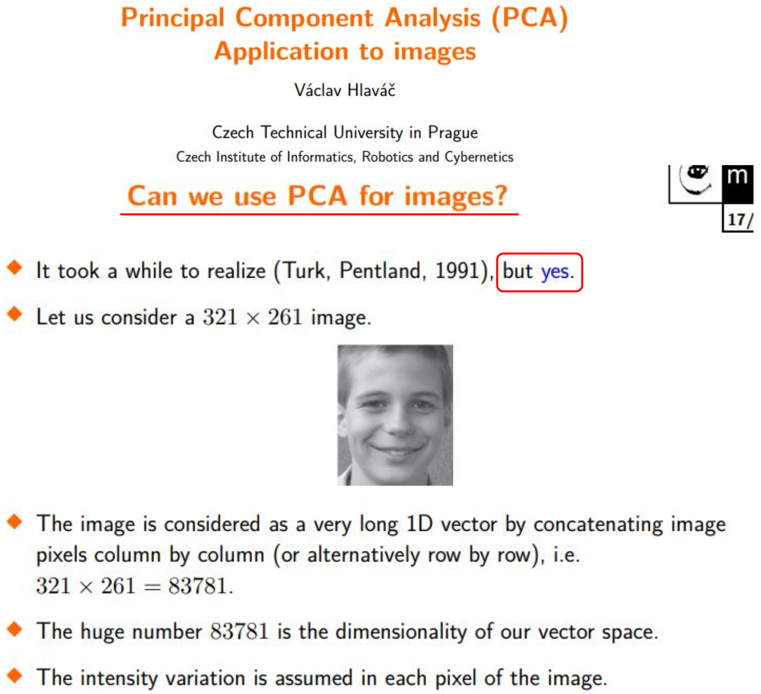
Principal Component Analysis (PCA)

1. Quick Understanding and Purpose
2. Visualizations and Metrics Received
3. How to further use data?

Initial Research and Understanding of PCA

- For high dimensional data, PCA is a method used to reduce the number of variables in data by extracting the important features from a large pool.
- PCA combines variables that are highly correlated together and form groups called ‘principal components’ that accounts for most variance in data
- Helps avoid overfitting by focusing on principal components instead of learning from non-important features
    - ‘Denoising’
- In our dataset, find features (areas or pixels) that are most important in determining if it the melt pool will result in a good or bad part

Using PCA on Images



Principal Component Analysis (PCA)
Application to images

Václav Hlaváč

Czech Technical University in Prague
Czech Institute of Informatics, Robotics and Cybernetics

Can we use PCA for images?

17/

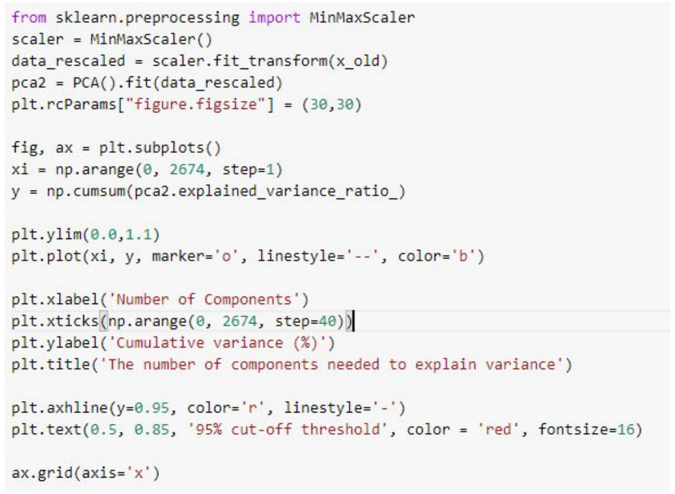
- It took a while to realize (Turk, Pentland, 1991), but yes.
- Let us consider a 321×261 image.

- The image is considered as a very long 1D vector by concatenating image pixels column by column (or alternatively row by row), i.e. $321 \times 261 = 83781$.
- The huge number 83781 is the dimensionality of our vector space.
- The intensity variation is assumed in each pixel of the image.

- First, to know that it is valid to use PCA on images, I read PCA Application to images by Dr. Hlavac from Czech Technical University in Prague
- Asserts that images can be used in PCA, as they can be converted to a one-dimensional vector by row-by-row or column-by-column concatenation
- Like our dataset, a grayscale image is a matrix of values with pixels corresponding to a value (the image on the slide represents intensity in the photo, our dataset a pixel represents a corresponding temperature)

Achieving a Desirable Variance

Code for Graph on Next Slide



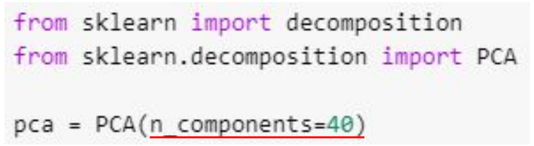
```python
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
data_rescaled = scaler.fit_transform(x_old)
pca2 = PCA().fit(data_rescaled)
plt.rcParams["figure.figsize"] = (30,30)

fig, ax = plt.subplots()
xi = np.arange(0, 2674, step=1)
y = np.cumsum(pca2.explained_variance_ratio_)

plt.ylim(0.0,1.1)
plt.plot(xi, y, marker='o', linestyle='--', color='b')

plt.xlabel('Number of Components')
plt.xticks(np.arange(0, 2674, step=40))
plt.ylabel('Cumulative variance (%)')
plt.title('The number of components needed to explain variance')

plt.axhline(y=0.95, color='r', linestyle='-')
plt.text(0.5, 0.85, '95% cut-off threshold', color = 'red', fontsize=16)

ax.grid(axis='x')
```

- PCA wants to account for the most variability possible in the dataset so that you can get unique features of the dataset
    - Choosing a number of components is important for PCA because it can yield a certain of variance
    - 95% variance seems to be a common for PCA models
- Instead of manually picking components, we can use visualization to figure out how many components to pick

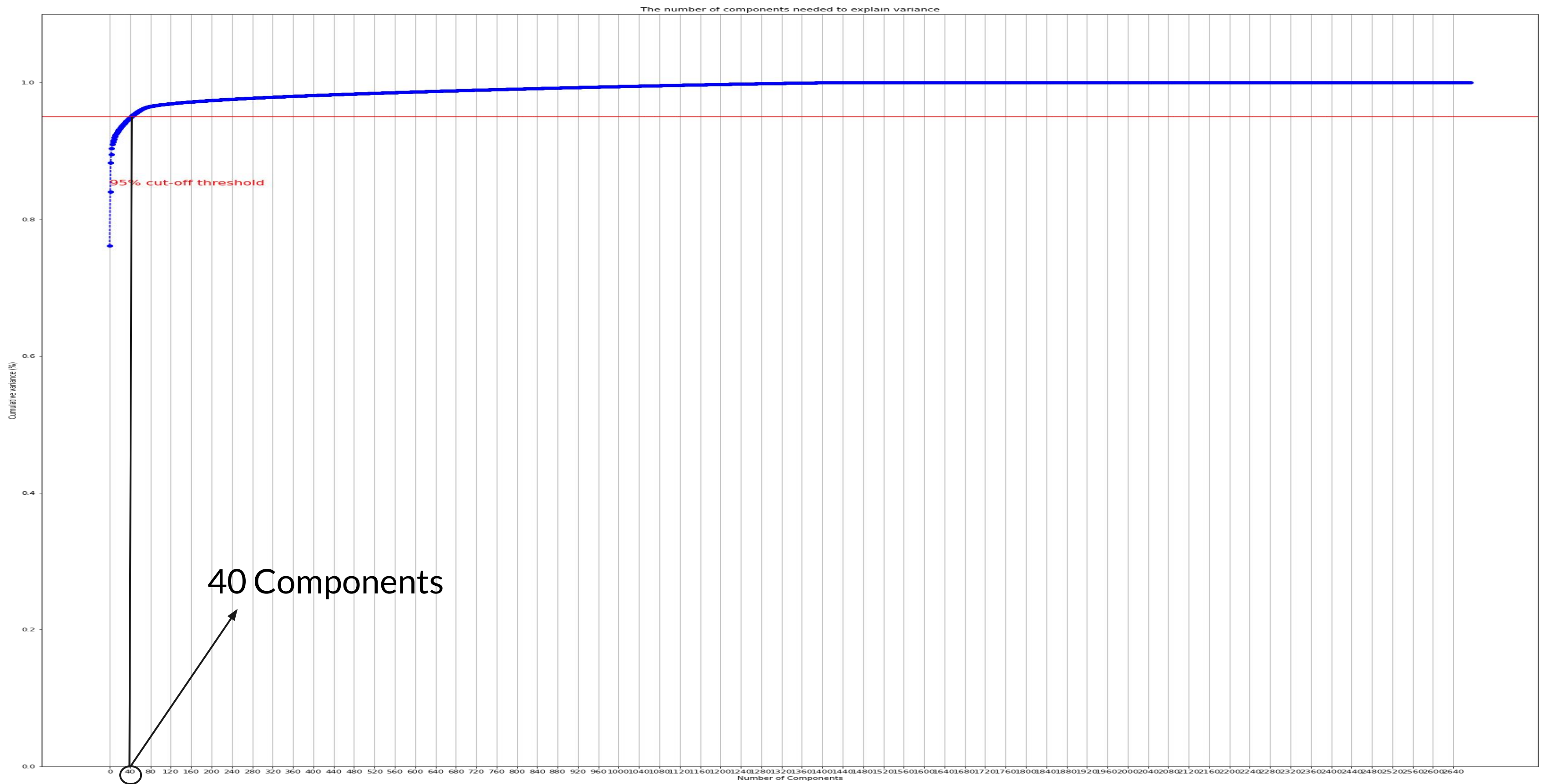
Variance Formula

$$S^2 = \frac{\sum (x_i - \bar{x})^2}{n - 1}$$

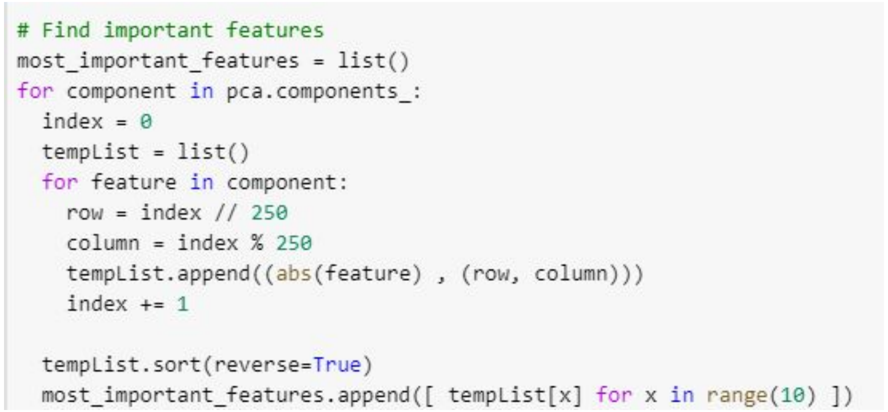
Choosing Components



```python
from sklearn import decomposition
from sklearn.decomposition import PCA

pca = PCA(n_components=40)
```


The number of components needed to explain variance
95% cut-off threshold
40 Components
Cumulative variance (%)
Number of Components

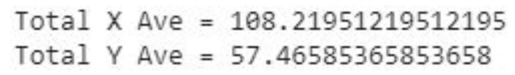
Finding Important Features



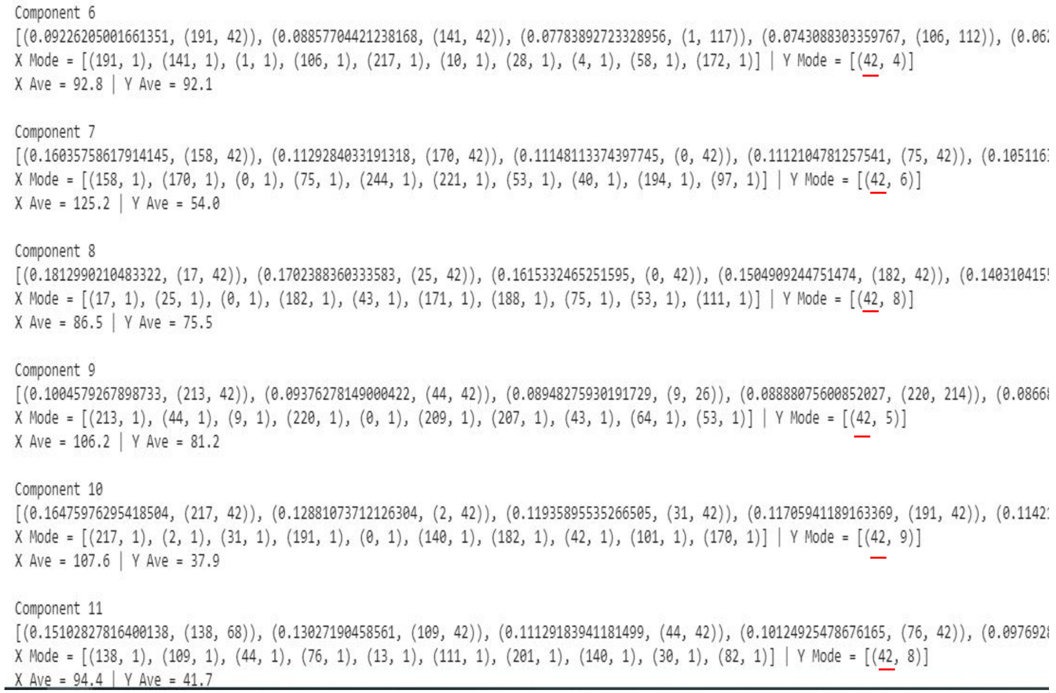
```
# Find important features
most_important_features = list()
for component in pca.components_:
  index = 0
  tempList = list()
  for feature in component:
    row = index // 250
    column = index % 250
    tempList.append((abs(feature) , (row, column)))
    index += 1

  tempList.sort(reverse=True)
  most_important_features.append([ tempList[x] for x in range(10) ])
```

- For each component, every feature has a magnitude of its corresponding values of its eigenvector
  - Bigger the magnitude, the more important it is
- To find these important components
  - Take the absolute value to get a magnitude
  - Sort by largest to smallest
  - Resulting in most important features being at top of list
- Added a (row, column) to see side-by-side how important each pixel is
- Took the top 10 most influential pixels for each component

Summary of PCA Results



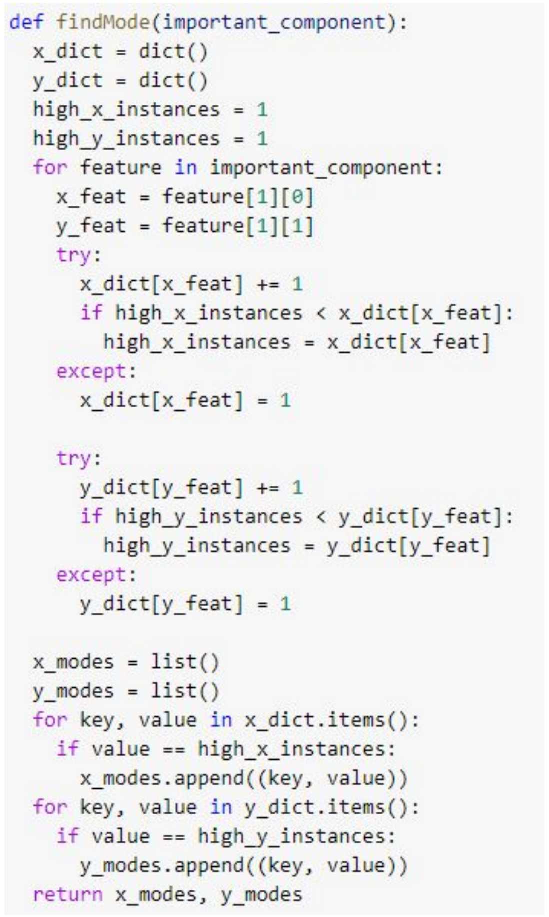
```
Total X Ave = 108.21951219512195
Total Y Ave = 57.46585365853658
```



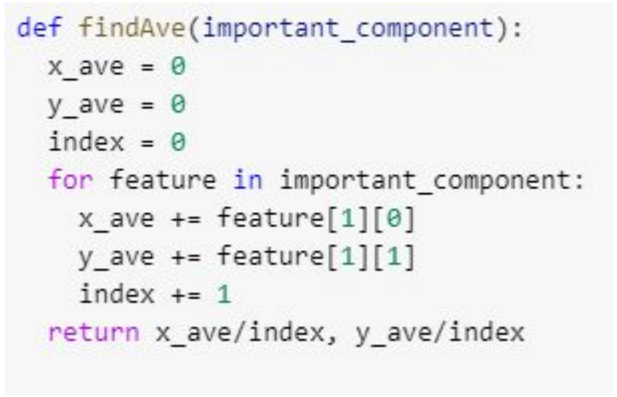
```
Component 6
[(0.09226205001661351, (191, 42)), (0.08857704421238168, (141, 42)), (0.07783892723328956, (1, 117)), (0.0743088303359767, (106, 112)), (0.06
X Mode = [(191, 1), (141, 1), (1, 1), (106, 1), (217, 1), (10, 1), (28, 1), (4, 1), (58, 1), (172, 1)] | Y Mode = [(42, 4)]
X Ave = 92.8 | Y Ave = 92.1

Component 7
[(0.16035758617914145, (158, 42)), (0.1129284033191318, (170, 42)), (0.11148113374397745, (0, 42)), (0.1112104781257541, (75, 42)), (0.105116
X Mode = [(158, 1), (170, 1), (0, 1), (75, 1), (244, 1), (221, 1), (53, 1), (40, 1), (194, 1), (97, 1)] | Y Mode = [(42, 6)]
X Ave = 125.2 | Y Ave = 54.0

Component 8
[(0.1812990210483322, (17, 42)), (0.1702388360333583, (25, 42)), (0.1615332465251595, (0, 42)), (0.1504909244751474, (182, 42)), (0.140310415
X Mode = [(17, 1), (25, 1), (0, 1), (182, 1), (43, 1), (171, 1), (188, 1), (75, 1), (53, 1), (111, 1)] | Y Mode = [(42, 8)]
X Ave = 86.5 | Y Ave = 75.5

Component 9
[(0.1004579267898733, (213, 42)), (0.09376278149000422, (44, 42)), (0.08948275930191729, (9, 26)), (0.08888075600852027, (220, 214)), (0.0866
X Mode = [(213, 1), (44, 1), (9, 1), (220, 1), (0, 1), (209, 1), (207, 1), (43, 1), (64, 1), (53, 1)] | Y Mode = [(42, 5)]
X Ave = 106.2 | Y Ave = 81.2

Component 10
[(0.16475976295418504, (217, 42)), (0.12881073712126304, (2, 42)), (0.11935895535266505, (31, 42)), (0.11705941189163369, (191, 42)), (0.1142
X Mode = [(217, 1), (2, 1), (31, 1), (191, 1), (0, 1), (140, 1), (182, 1), (42, 1), (101, 1), (170, 1)] | Y Mode = [(42, 9)]
X Ave = 107.6 | Y Ave = 37.9

Component 11
[(0.1510282781640138, (138, 68)), (0.13027190458561, (109, 42)), (0.11129183941181499, (44, 42)), (0.10124925478676165, (76, 42)), (0.097692
X Mode = [(138, 1), (109, 1), (44, 1), (76, 1), (13, 1), (111, 1), (201, 1), (140, 1), (30, 1), (82, 1)] | Y Mode = [(42, 8)]
X Ave = 94.4 | Y Ave = 41.7
```

- For Modes (position, instances)
  - Position is either its x or y position
  - Instances is how many time that position occurred
- Can see the magnitude of an eigenvector (importance) next to its row, column coordinates
- Took x and y averages of top 10 most important features
- Took averages of the 40 componentes x and y averages
- y = 42, showed up a lot as a mode

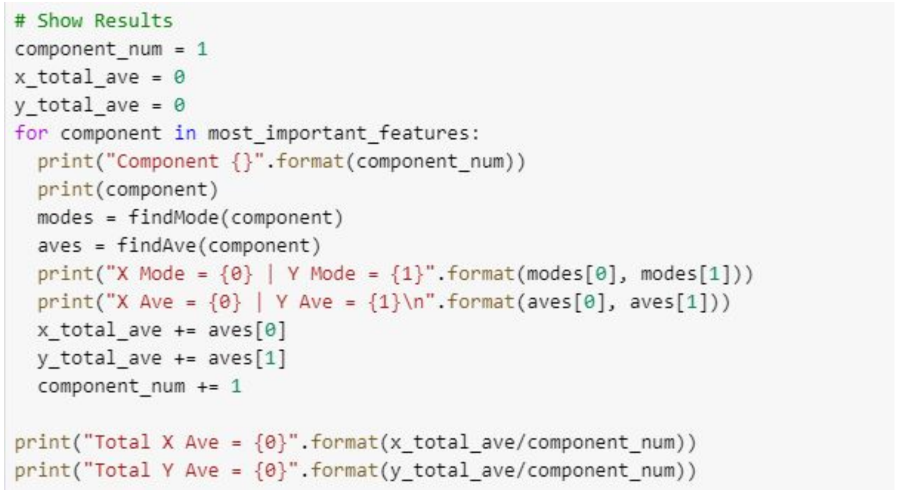
Code for Summary of PCA Results



```python
def findMode(important_component):
  x_dict = dict()
  y_dict = dict()
  high_x_instances = 1
  high_y_instances = 1
  for feature in important_component:
    x_feat = feature[1][0]
    y_feat = feature[1][1]
    try:
      x_dict[x_feat] += 1
      if high_x_instances < x_dict[x_feat]:
        high_x_instances = x_dict[x_feat]
    except:
      x_dict[x_feat] = 1

    try:
      y_dict[y_feat] += 1
      if high_y_instances < y_dict[y_feat]:
        high_y_instances = y_dict[y_feat]
    except:
      y_dict[y_feat] = 1

  x_modes = list()
  y_modes = list()
  for key, value in x_dict.items():
    if value == high_x_instances:
      x_modes.append((key, value))
  for key, value in y_dict.items():
    if value == high_y_instances:
      y_modes.append((key, value))
  return x_modes, y_modes
```



```python
def findAve(important_component):
  x_ave = 0
  y_ave = 0
  index = 0
  for feature in important_component:
    x_ave += feature[1][0]
    y_ave += feature[1][1]
    index += 1
  return x_ave/index, y_ave/index
```



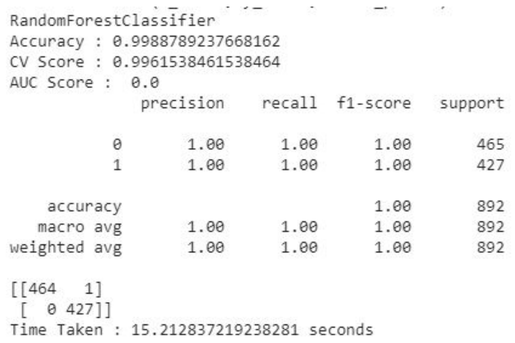
```python
# Show Results
component_num = 1
x_total_ave = 0
y_total_ave = 0
for component in most_important_features:
  print("Component {}".format(component_num))
  print(component)
  modes = findMode(component)
  aves = findAve(component)
  print("X Mode = {0} | Y Mode = {1}".format(modes[0], modes[1]))
  print("X Ave = {0} | Y Ave = {1}\n".format(aves[0], aves[1]))
  x_total_ave += aves[0]
  y_total_ave += aves[1]
  component_num += 1

print("Total X Ave = {0}".format(x_total_ave/component_num))
print("Total Y Ave = {0}".format(y_total_ave/component_num))
```

Random Forest Exploration

1. Why further explore Random Forest Classifier
2. Understanding of Vectorizing a Matrix
3. More testing and results of Random Forest for various parameters

Why further explore Random Forest Classifier?

- Had very high accuracy (above 99% accuracy in classifying)
- Had very fast time compared to other ensemble methods
- Test to see if it can if the first trial was dumb luck or if it can be repeated
- Address whether vectorizing a matrix would lose relationships from the data

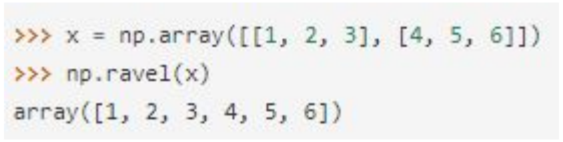


```
RandomForestClassifier
Accuracy : 0.9988789237668162
CV Score : 0.9961538461538464
AUC Score :  0.0
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       465
           1       1.00      1.00      1.00       427

    accuracy                           1.00       892
   macro avg       1.00      1.00      1.00       892
weighted avg       1.00      1.00      1.00       892

[[464   1]
 [  0 427]]
Time Taken : 15.212837219238281 seconds
```

Vectorizing a Matrix Keeps Order and Relationship

$$data = \begin{bmatrix} (0,0) & \ldots & (0,249) \\ \vdots & \ddots & \\ (249,0) & & (249,249) \end{bmatrix} \xrightarrow{\text{Flatten}} vector = [\underset{\substack{\uparrow \\ (0,0)}}{0} \quad \ldots \quad \underset{\substack{\uparrow \\ (0,249)}}{249} \quad \ldots \quad \underset{\substack{\uparrow \\ (249,0)}}{62250} \quad \ldots \quad \underset{\substack{\uparrow \\ (249,249)}}{62499}]$$

Example Python Code



```
>>> x = np.array([[1, 2, 3], [4, 5, 6]])
>>> np.ravel(x)
array([1, 2, 3, 4, 5, 6])
```

Conversion Formulas

From Vector to Matrix

- Row = Vector_i / 250
- Column = Vector_i % 250
- In Matrix: (Row, Column) Order

From Matrix to Vector

- Vector_i = Row * 250 + Column

Testing Random Forest Classifier

Procedure

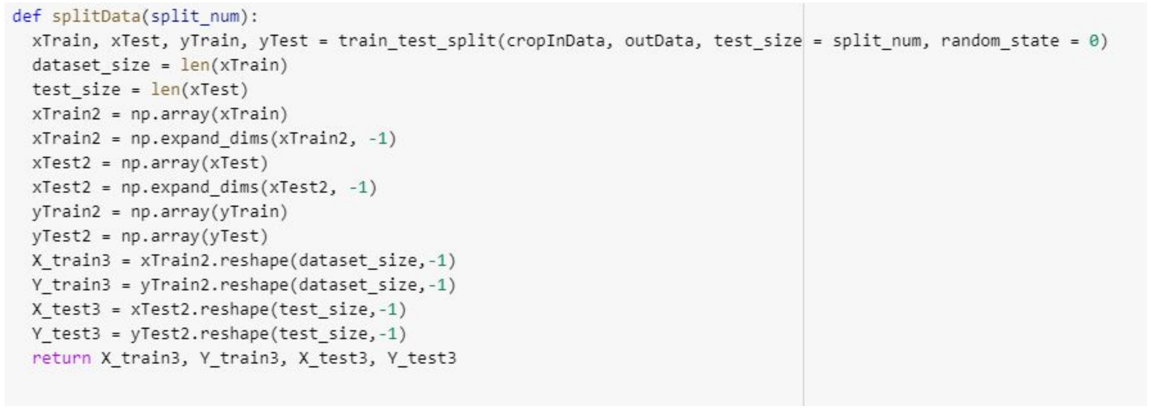
- 100 Total Trials
- 25 Trials each for
  - 50/50 Train/Test Split
  - 60/40 Train/Test Split
  - 70/30 Train/Test Split
  - 80/20 Train/Test Split
- Every trial a new data split will be calculated to ensure random splits

Outcomes

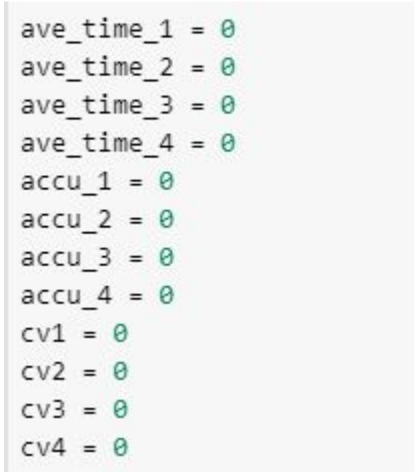
For each type of train/test split over 25 trials:

1. Average Accuracy Score
2. Average Cross Validation Score
3. Average Time Taken per Trial

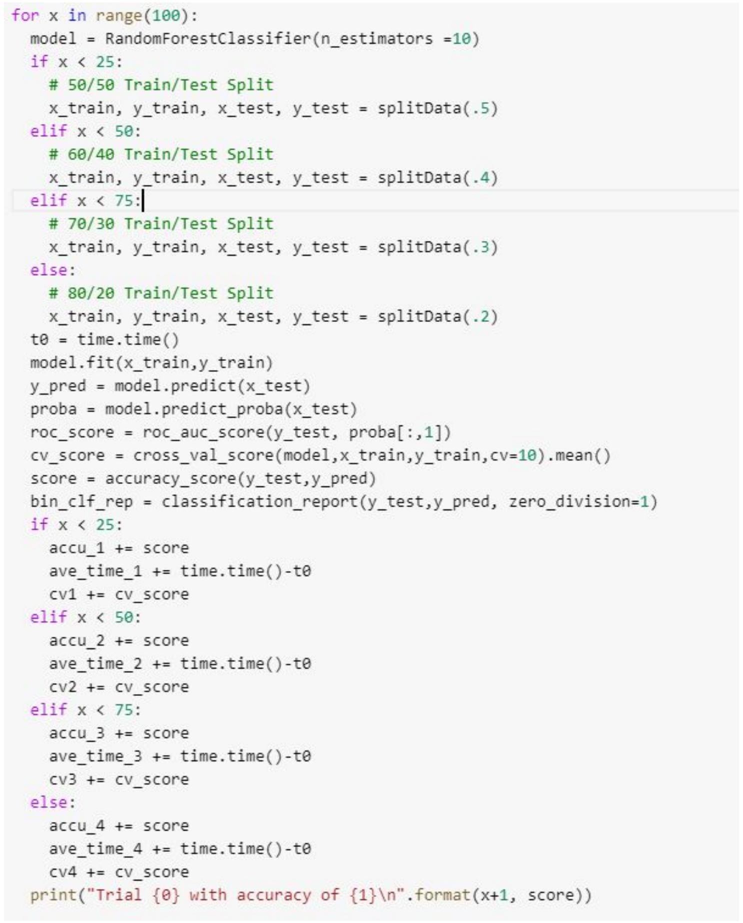
Code for Testing Random Forest Classifier



```
def splitData(split_num):
  xTrain, xTest, yTrain, yTest = train_test_split(cropInData, outData, test_size = split_num, random_state = 0)
  dataset_size = len(xTrain)
  test_size = len(xTest)
  xTrain2 = np.array(xTrain)
  xTrain2 = np.expand_dims(xTrain2, -1)
  xTest2 = np.array(xTest)
  xTest2 = np.expand_dims(xTest2, -1)
  yTrain2 = np.array(yTrain)
  yTest2 = np.array(yTest)
  X_train3 = xTrain2.reshape(dataset_size,-1)
  Y_train3 = yTrain2.reshape(dataset_size,-1)
  X_test3 = xTest2.reshape(test_size,-1)
  Y_test3 = yTest2.reshape(test_size,-1)
  return X_train3, Y_train3, X_test3, Y_test3
```

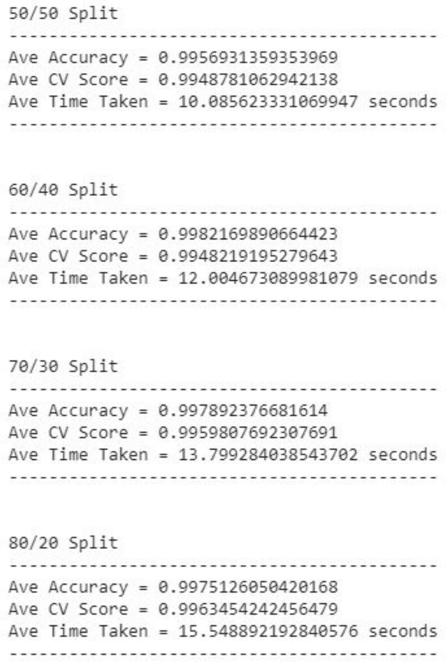


```
ave_time_1 = 0
ave_time_2 = 0
ave_time_3 = 0
ave_time_4 = 0
accu_1 = 0
accu_2 = 0
accu_3 = 0
accu_4 = 0
cv1 = 0
cv2 = 0
cv3 = 0
cv4 = 0
```



```
for x in range(100):
  model = RandomForestClassifier(n_estimators =10)
  if x < 25:
    # 50/50 Train/Test Split
    x_train, y_train, x_test, y_test = splitData(.5)
  elif x < 50:
    # 60/40 Train/Test Split
    x_train, y_train, x_test, y_test = splitData(.4)
  elif x < 75:
    # 70/30 Train/Test Split
    x_train, y_train, x_test, y_test = splitData(.3)
  else:
    # 80/20 Train/Test Split
    x_train, y_train, x_test, y_test = splitData(.2)
  t0 = time.time()
  model.fit(x_train,y_train)
  y_pred = model.predict(x_test)
  proba = model.predict_proba(x_test)
  roc_score = roc_auc_score(y_test, proba[:,1])
  cv_score = cross_val_score(model,x_train,y_train,cv=10).mean()
  score = accuracy_score(y_test,y_pred)
  bin_clf_rep = classification_report(y_test,y_pred, zero_division=1)
  if x < 25:
    accu_1 += score
    ave_time_1 += time.time()-t0
    cv1 += cv_score
  elif x < 50:
    accu_2 += score
    ave_time_2 += time.time()-t0
    cv2 += cv_score
  elif x < 75:
    accu_3 += score
    ave_time_3 += time.time()-t0
    cv3 += cv_score
  else:
    accu_4 += score
    ave_time_4 += time.time()-t0
    cv4 += cv_score
  print("Trial {0} with accuracy of {1}\n".format(x+1, score))
```

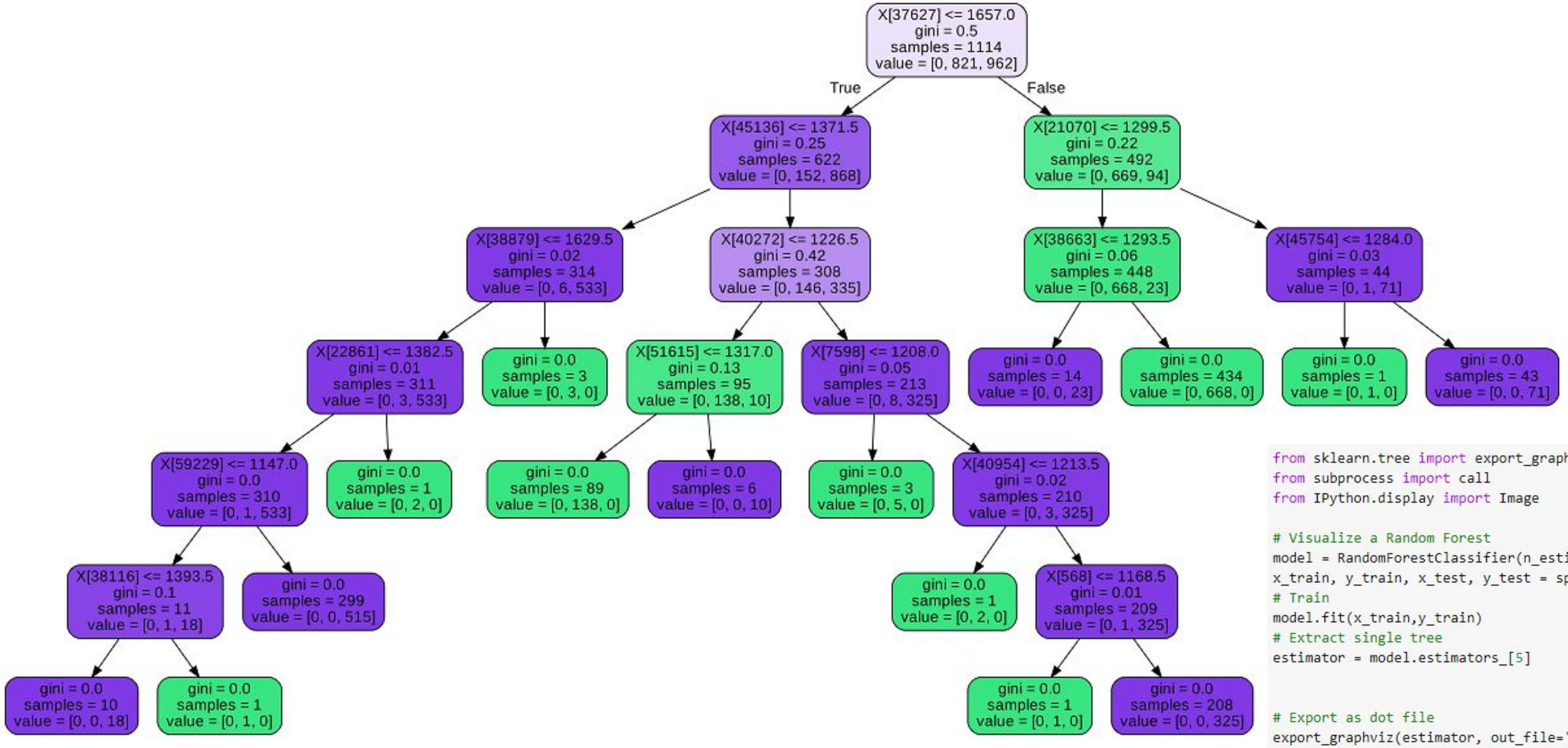
Results for Testing Random Forest

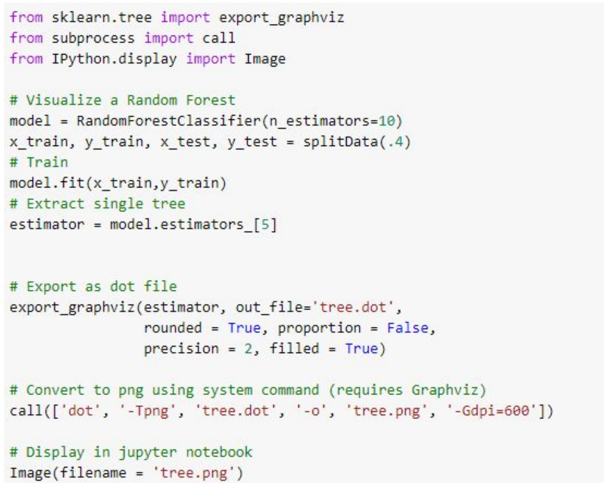


```
50/50 Split
-------------------------------------------
Ave Accuracy = 0.9956931359353969
Ave CV Score = 0.9948781062942138
Ave Time Taken = 10.085623331069947 seconds
-------------------------------------------


60/40 Split
-------------------------------------------
Ave Accuracy = 0.9982169890664423
Ave CV Score = 0.9948219195279643
Ave Time Taken = 12.004673089981079 seconds
-------------------------------------------


70/30 Split
-------------------------------------------
Ave Accuracy = 0.997892376681614
Ave CV Score = 0.9959807692307691
Ave Time Taken = 13.799284038543702 seconds
-------------------------------------------


80/20 Split
-------------------------------------------
Ave Accuracy = 0.9975126050420168
Ave CV Score = 0.9963454242456479
Ave Time Taken = 15.548892192840576 seconds
-------------------------------------------
```

- Highest average accuracy was 60/40 Train/Test split
  - Accuracy of 99.821...%
- Lowest average accuracy was 50/50 Train/Test split
  - Accuracy of 99.569...%
- Time went up with the greater the split towards training
  - Logically, this makes sense as the model has more to data values to train on, the longer it will take

Visualizing a Decision Tree in a Random Forest Classifier





```
from sklearn.tree import export_graphviz
from subprocess import call
from IPython.display import Image

# Visualize a Random Forest
model = RandomForestClassifier(n_estimators=10)
x_train, y_train, x_test, y_test = splitData(.4)
# Train
model.fit(x_train,y_train)
# Extract single tree
estimator = model.estimators_[5]


# Export as dot file
export_graphviz(estimator, out_file='tree.dot',
                rounded = True, proportion = False,
                precision = 2, filled = True)

# Convert to png using system command (requires Graphviz)
call(['dot', '-Tpng', 'tree.dot', '-o', 'tree.png', '-Gdpi=600'])

# Display in jupyter notebook
Image(filename = 'tree.png')
```

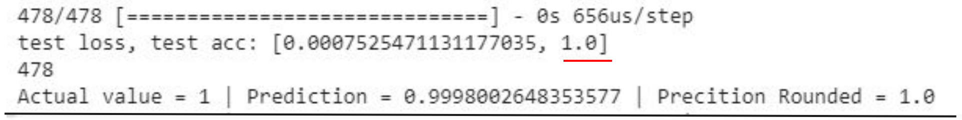
Trying More Neural Networks

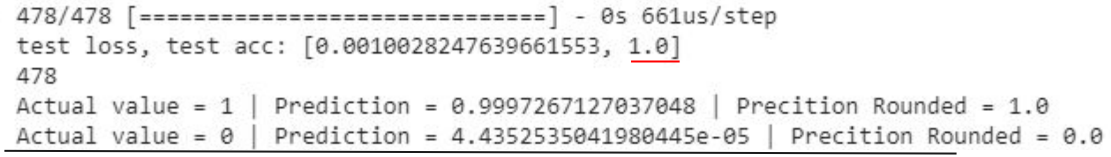
1. Overview of what parameters and architectures I was using
2. Overview of code and results of a currently successful model
3. Need for more testing

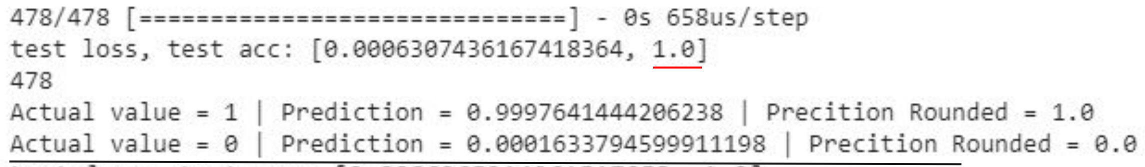
Trying different Models Tried

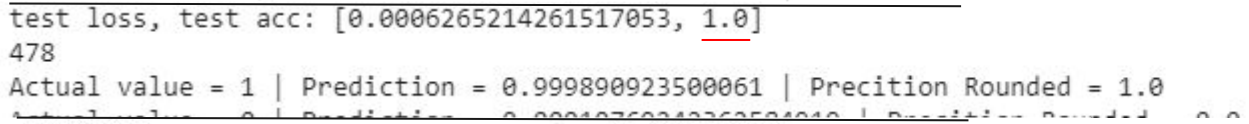
- A lot of time was spent editing different parameters from last weeks neural network (ave 79% test accuracy over 10 trials)
- Different parameters included:
  - Epochs
  - Batch Size
  - Learning Rate
  - Optimizers
  - Activation Functions
  - Many Hidden Layers
  - Few Hidden Layers
  - Different Types of Layer
  - Batch Normalization
  - Dropout Rates

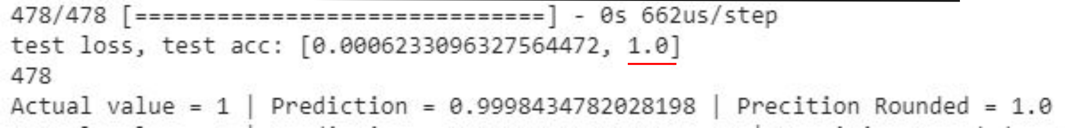
A Promising Result

1.

```
478/478 [==============================] - 0s 656us/step
test loss, test acc: [0.0007525471131177035, 1.0]
478
Actual value = 1 | Prediction = 0.9998002648353577 | Precition Rounded = 1.0
```

2.

```
478/478 [==============================] - 0s 661us/step
test loss, test acc: [0.0010028247639661553, 1.0]
478
Actual value = 1 | Prediction = 0.9997267127037048 | Precition Rounded = 1.0
Actual value = 0 | Prediction = 4.4352535041980445e-05 | Precition Rounded = 0.0
```

3.

```
478/478 [==============================] - 0s 658us/step
test loss, test acc: [0.0006307436167418364, 1.0]
478
Actual value = 1 | Prediction = 0.9997641444206238 | Precition Rounded = 1.0
Actual value = 0 | Prediction = 0.00016337945999111198 | Precition Rounded = 0.0
```

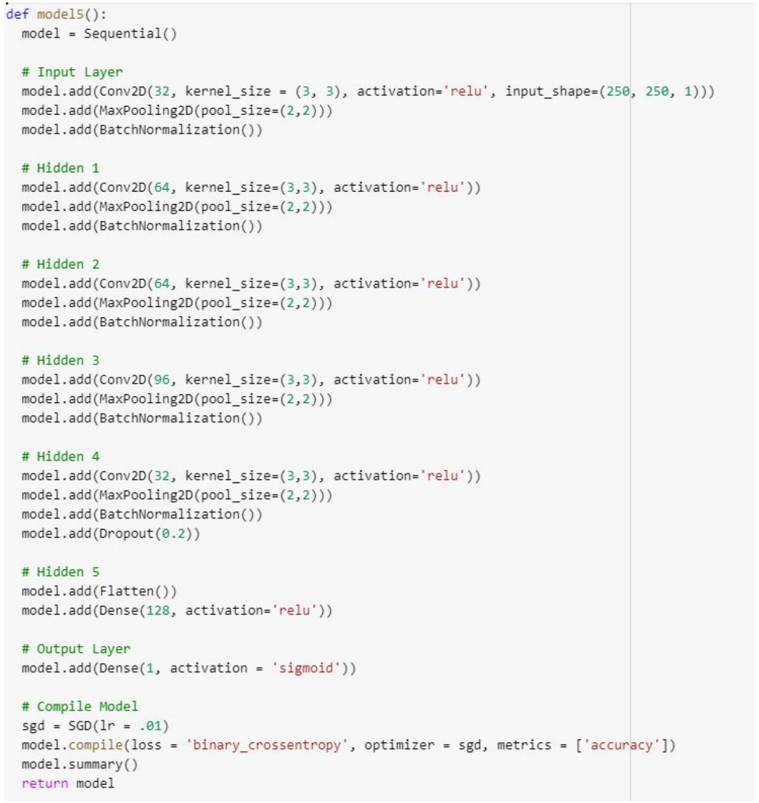
4.

```
test loss, test acc: [0.0006265214261517053, 1.0]
478
Actual value = 1 | Prediction = 0.999890923500061 | Precition Rounded = 1.0
[illegible]
```

5.

```
478/478 [==============================] - 0s 662us/step
test loss, test acc: [0.0006233096327564472, 1.0]
478
Actual value = 1 | Prediction = 0.9998434782028198 | Precition Rounded = 1.0
[illegible]
```

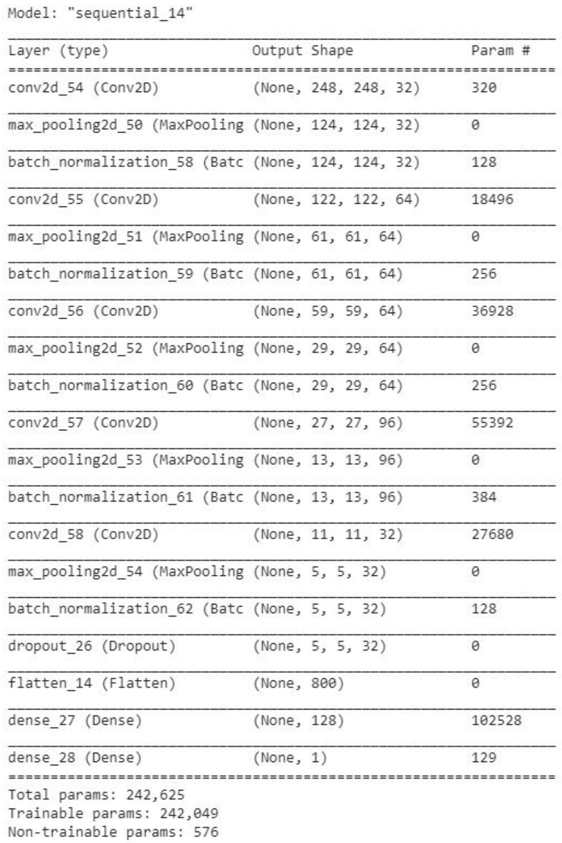
5 Trials of

- 70/30 Train/Test Split
  - Every trial had a different split
- Every trail had a test accuracy of 100%
- 20 Epochs
- Batch Size of 32

The Model’s Architecture / Code



```
def model5():
  model = Sequential()

  # Input Layer
  model.add(Conv2D(32, kernel_size = (3, 3), activation='relu', input_shape=(250, 250, 1)))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(BatchNormalization())

  # Hidden 1
  model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(BatchNormalization())

  # Hidden 2
  model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(BatchNormalization())

  # Hidden 3
  model.add(Conv2D(96, kernel_size=(3,3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(BatchNormalization())

  # Hidden 4
  model.add(Conv2D(32, kernel_size=(3,3), activation='relu'))
  model.add(MaxPooling2D(pool_size=(2,2)))
  model.add(BatchNormalization())
  model.add(Dropout(0.2))

  # Hidden 5
  model.add(Flatten())
  model.add(Dense(128, activation='relu'))

  # Output Layer
  model.add(Dense(1, activation = 'sigmoid'))

  # Compile Model
  sgd = SGD(lr = .01)
  model.compile(loss = 'binary_crossentropy', optimizer = sgd, metrics = ['accuracy'])
  model.summary()
  return model
```



```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_54 (Conv2D)           (None, 248, 248, 32)      320
_________________________________________________________________
max_pooling2d_50 (MaxPooling (None, 124, 124, 32)      0
_________________________________________________________________
batch_normalization_58 (Batc (None, 124, 124, 32)      128
_________________________________________________________________
conv2d_55 (Conv2D)           (None, 122, 122, 64)      18496
_________________________________________________________________
max_pooling2d_51 (MaxPooling (None, 61, 61, 64)        0
_________________________________________________________________
batch_normalization_59 (Batc (None, 61, 61, 64)        256
_________________________________________________________________
conv2d_56 (Conv2D)           (None, 59, 59, 64)        36928
_________________________________________________________________
max_pooling2d_52 (MaxPooling (None, 29, 29, 64)        0
_________________________________________________________________
batch_normalization_60 (Batc (None, 29, 29, 64)        256
_________________________________________________________________
conv2d_57 (Conv2D)           (None, 27, 27, 96)        55392
_________________________________________________________________
max_pooling2d_53 (MaxPooling (None, 13, 13, 96)        0
_________________________________________________________________
batch_normalization_61 (Batc (None, 13, 13, 96)        384
_________________________________________________________________
conv2d_58 (Conv2D)           (None, 11, 11, 32)        27680
_________________________________________________________________
max_pooling2d_54 (MaxPooling (None, 5, 5, 32)          0
_________________________________________________________________
batch_normalization_62 (Batc (None, 5, 5, 32)          128
_________________________________________________________________
dropout_26 (Dropout)         (None, 5, 5, 32)          0
_________________________________________________________________
flatten_14 (Flatten)         (None, 800)               0
_________________________________________________________________
dense_27 (Dense)             (None, 128)               102528
_________________________________________________________________
dense_28 (Dense)             (None, 1)                 129
=================================================================
Total params: 242,625
Trainable params: 242,049
Non-trainable params: 576
```

- Uses ‘blocks’ of Conv2D, MaxPooling2D, and Batch Normalization
- Finally flattens to a dense layer
- Uses SGD optimizer
- Every layer but the final layer uses relu
- Final layer uses sigmoid

Need to Further Test This Model

- Got first promising results for this model yesterday (6/24)
- Need to test:
  - Different train/test split ratios
  - More Epoch Sizes
    - Originally tested 5 epochs, that had lower accuracy scores
    - 20 Epochs was the text value tested, which yielded very good accuracy scores
  - Batch Size
- Essentially, test this model more thoroughly like the Random Forest Classifier

Future Steps and Goals

Also Google Colab has Corgi Mode, it's great =)

- Test the new neural network model more thorough and give results
- What to do with PCA data / results?
- Suffering from Black Box Testing when construction Neural Networks
  - Advice?
- Ultimate Goal of Project Clarification
  - Should the final model be a neural network, or whatever ascertains a high classification accuracy?
- Any obvious next steps?