Final Project

SYS 6018 - Data Mining
Holden Bruce
hab6xf

Some major points of feedback I received from Part 1 that I implemented in this final project submission:

- Improved Exploratory Data Analysis
  - more questions being asked of the data before diving in
  - better labels for plots
- Implementing GridSearchCV
  - From Scott: "While you don't *have* to use these capabilities [referring to GridSearchCV and KFold] for part 2 of the project, my personal feeling is that one coming out of the ISLR level (but for Python) *absolutely SHOULD* have these tools as their go to's"
  - I first implemented GridSearchCV for finding the optimal hyperparameters through the use of param_grid for both random forest and support vector machines. Then I replaced the train_test_split with KFold, since Scott argued that this method is better in every way.
- Implementing KFolds instead of train_test_split() from scikit-learn https://datascience.stackexchange.com/questions/52632/cross-validation-vs-train-validate-test https://towardsdatascience.com/complete-guide-to-pythons-cross-validation-with-examples-a9676b5cac12 https://towardsdatascience.com/train-test-split-and-cross-validation-in-python-80b61beca4b6 https://towardsdatascience.com/5-reasons-why-you-should-use-cross-validation-in-your-data-science-project-8163311a1e79#:~:text=The%20training%20set%20is%20used,do%20more%20than%20one% ^ these helped quite a bit as I was exploring KFolds and learning how to implement it in place of train_test_split.
- Labeling Blue Tarps as 'positive' instead of as 'negative'
  - in Part 1 I had this designation flipped which, as Scott rightly pointed out, is pretty confusing to whoever is reviewing my findings since it goes against the convention of having 1 (positive) being associated with the thing you are looking for and 0 typically representing the opposite.
- Actually considered the uncertainty of scores in the selection process.

All the libraries and packages used in this project

In [433…

```
import numpy as np
import matplotlib.pyplot as plt
from itertools import cycle

from sklearn import svm, datasets
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import train_test_split
from sklearn.model_selection import KFold
```

```
from sklearn.preprocessing import label_binarize
from sklearn.multiclass import OneVsRestClassifier
from scipy import interp
from sklearn.metrics import roc_auc_score

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn import neighbors
from sklearn.metrics import confusion_matrix, classification_report,accuracy_sco
from sklearn.metrics import precision_recall_fscore_support
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split


from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.linear_model import LogisticRegression

from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import RandomizedSearchCV

from sklearn.utils import resample # downsample the dataset
from sklearn import preprocessing # scale and center data
from sklearn.svm import SVC # this will make a support vector machine for classi
from sklearn.metrics import plot_confusion_matrix # draws a confusion matrix
from sklearn.model_selection import GridSearchCV # this will do cross validation

import webcolors
from mpl_toolkits import mplot3d
```

Another piece of feedback I got from Scott after the submission of Part 1 was that I could improve the readability of some of the tables and graphs I had. So this next section sets the precision and general format for data output in this project. I've chosen to specify the precision of decimals returned in pandas to 3.

In [74]:
```
pd.set_option('precision', 3) # number precision for pandas
pd.set_option('display.max_rows', 12)
pd.set_option('display.max_columns', 12)
pd.set_option('display.float_format', '{:20,.4f}'.format) # get rid of scientifi
plt.style.use('seaborn') # pretty matplotlib plots
```

In [75]:
```
pixels = pd.read_csv('HaitiPixels.csv', na_values=["?"])
holdout = pd.read_csv('concat_data.csv', na_values=["?"])
# see the starting.py submission for how I created the holdout dataset
```

In [76]:
```
pixels.head()
```

Out[76]:

	Class	Red	Green	Blue
0	Vegetation	64	67	50
1	Vegetation	64	67	50
2	Vegetation	64	66	49

	Class	Red	Green	Blue
3	Vegetation	75	82	53
4	Vegetation	74	82	54

In [77]:
```
pixels.info()
# Class is type object
# Red, Green, Blue are all type int64
# This is expected. The Red, Green, Blue columns represent the RGB color scheme
# while the Class column represents the Classification of the object or pixel
# from the original picture that his dataset was developed from.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 63241 entries, 0 to 63240
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   Class   63241 non-null  object
 1   Red     63241 non-null  int64 
 2   Green   63241 non-null  int64 
 3   Blue    63241 non-null  int64 
dtypes: int64(3), object(1)
memory usage: 1.9+ MB
```

In [78]:
```
pixels['Class'].unique()
```

Out[78]:
```
array(['Vegetation', 'Soil', 'Rooftop', 'Various Non-Tarp', 'Blue Tarp'],
      dtype=object)
```

We see here that there are 5 classifications in the Class column. However, I am going to approach this as a binary classification problem instead of as a multiclass problem, so a bit lower I will map the classification 'Blue Tarp' to 1 and the other 4 classifications to 0.

In [79]:
```
holdout.head()
```

Out[79]:

	Red	Green	Blue	Class
0	104	89	63	0
1	101	80	60	0
2	103	87	69	0
3	107	93	72	0
4	109	99	68	0

In [80]:
```
holdout.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2008623 entries, 0 to 2008622
Data columns (total 4 columns):
 #   Column  Dtype
---  ------  -----
 0   Red     int64
 1   Green   int64
 2   Blue    int64
 3   Class   int64
dtypes: int64(4)
memory usage: 61.3 MB
```

Unlike the pixels dataset, all 4 columns in the holdout dataset are type int64. This is expected since I already mapped the data to 0 and 1 (see starting.py file). For every file that was designated as "NON" or "NOT", I created a new column "Class" and populated every row with 0. For all the other files, I created a new column "Class" and populated every row with 1.

This next function takes in a dataset and splits it into X and y, where y is the "Class" column and X are the Red, Green, Blue columns. The if-statement near the bottom of the function will not apply to the holdout dataset because I already did the 0/1 mapping in the starting.py file, but it will apply to the pixels dataset and turn the 5 different classifications seen in the "Class" column (['Vegetation', 'Soil', 'Rooftop', 'Various Non-Tarp', 'Blue Tarp']) into a list of 0's and 1's where "Blue Tarp" is mapped to 1 and the rest are mapped to 0. The reason why I have done this is that it makes it easier dealing with some of the models you will see below.

In [81]:
```
def load_data(dataset):

    #set X and y and then create train and test data from the dataset
    X = dataset.drop(['Class'], axis=1) #X is columns: Red, Green, Blue

    y = dataset.Class #Y is the Class column
    # if 'Class' in dataset.columns:
    #     y = dataset.Class #Y is the Class column
    # if 'tarp' in dataset.columns:
    #     y = dataset.tarp
    #this remapping turns Blue Tarp into classification=1 while every other choi
    #is mapped to 0...thus making it a binary

    #if y is still an object containing multiple categories, map those categorie
    #otherwise, don't do anything so that the X,y split from this function can b
    #used for other purposes
    if y.dtype != 'int64':
        dataset.Class = dataset.Class.astype('category')
        y = y.map({'Blue Tarp':1, 'Rooftop':0,'Soil':0,'Various Non-Tarp':0, 'Ve

        #rewrite the Class column in pixels with the new mapped version stored i
        pixels['Class'] = y

    return X,y
```

The pixels dataset has 63,241 observations. The holdout dataset has 2,008,623 observations. For now, we are only going to explore the pixels dataset. However, later on in this notebook I am going to call the load_data() function again, differentiating the pixels and holdout X and y values for a different purpose: to use the +2mil holdout dataset as our training data to be tested on the pixels dataset. The variables will be called X_pixels, y_pixels, X_holdout, and y_holdout.

But for now, I will just be using the pixels dataset and will only refer to them as X and y.

In [82]:
```
X,y = load_data(pixels)
```

Just making sure everything looks good:

In [83]:
```
X.shape, y.shape
```

Out[83]:
```
((63241, 3), (63241,))
```

In [84]:
```
X.head()
```

Out[84]:

	Red	Green	Blue
0	64	67	50
1	64	67	50
2	64	66	49
3	75	82	53
4	74	82	54

In [85]:
```
y.unique()
```

Out[85]:
```
array([0, 1])
```

Now I'm going to use webcolors to convert RGB to hexcode, then I will use the hexcode values for each row to populate a 3D scatterplot.

In [94]:
```
X_plot = X.copy()
print(X_plot)
```

```
       Red  Green  Blue
0       64     67    50
1       64     67    50
2       64     66    49
3       75     82    53
4       74     82    54
...    ...    ...   ...
63236  138    146   150
63237  134    141   152
63238  136    143   151
63239  132    139   149
63240  133    141   153

[63241 rows x 3 columns]
```

In [99]:
```
# going to use webcolors to convert RGB to hexcode
# conda install -c conda-forge webcolors
# import webcolors

# the three possible classes
classes = ['Red','Green','Blue']

# loop through each row of X
for row in range(len(X_plot)):
    red = 0 #temporary assignment of red
    green = 0 #temporary assignment of green
    blue = 0 #temporary assignment of blue

    # loop through the three columns in each row
    for color in classes:
        if color == 'Red':
            red = X_plot[color][row] #assign temporary variable red to the value
        if color == 'Green':
            green = X_plot[color][row] #assign temporary variable green to the v
        if color == 'Blue':
            blue = X_plot[color][row] #assign temporary variable blue to the val

            # add a new column 'hex' with the hex values calculated from the tem
```

```
            X_plot['hex'][row] = webcolors.rgb_to_hex((red,green,blue))

print(X_plot)
```

```
/Users/holdenbruce/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.
py:24: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
       Red  Green  Blue      hex
0       64     67    50  #404332
1       64     67    50  #404332
2       64     66    49  #404231
3       75     82    53  #4b5235
4       74     82    54  #4a5236
...    ...    ...   ...      ...
63236  138    146   150  #8a9296
63237  134    141   152  #868d98
63238  136    143   151  #888f97
63239  132    139   149  #848b95
63240  133    141   153  #858d99

[63241 rows x 4 columns]
```

In [100…

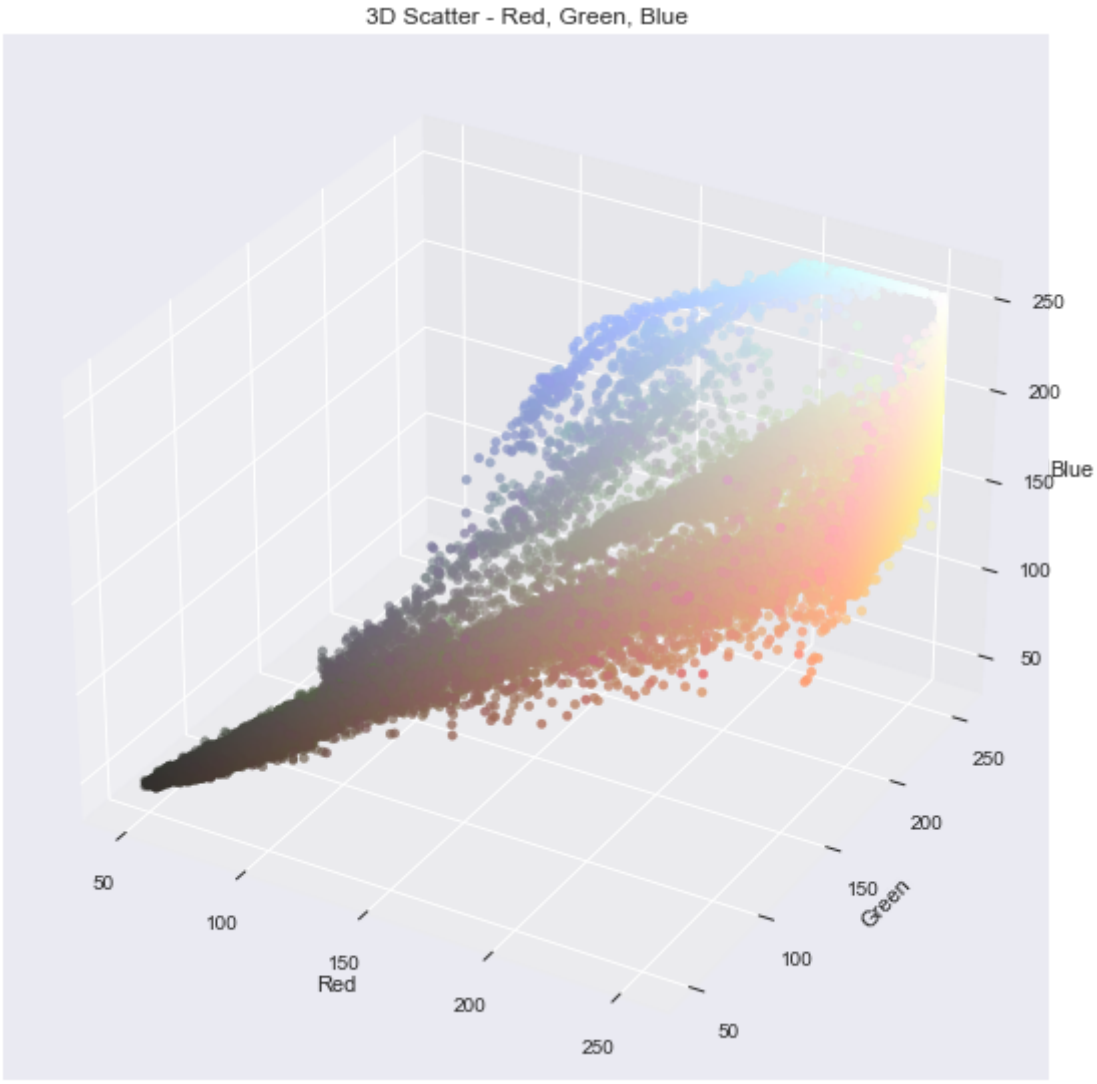
```
X_plot['hex']
```

Out[100…

```
0        #404332
1        #404332
2        #404231
3        #4b5235
4        #4a5236
          ...
63236    #8a9296
63237    #868d98
63238    #888f97
63239    #848b95
63240    #858d99
Name: hex, Length: 63241, dtype: object
```

In [101…

```
%matplotlib inline
plt.rcParams["figure.figsize"] = 12.8, 9.6

ax = plt.axes(projection='3d')
ax.scatter(np.reshape(X_plot['Red'], -1), np.reshape(X_plot['Green'], -1), np.re
ax.set_title('3D Scatter - Red, Green, Blue')
ax.set_xlabel('Red')
ax.set_ylabel('Green')
ax.set_zlabel('Blue')
plt.show()
```



Splitting the data into test and train sets (KFolds)

In Part 1 of the project I used train_test_split() to split the data into training data and testing data. But for Part 2, I will use KFold to split the data.

https://medium.com/@hjhuney/implementing-a-random-forest-classification-model-in-python-583891c99652

"Random forests tend to shine in scenarios where a model has a large number of features that individually have weak predicative power but much stronger power collectively."

Function I wrote for Part 1 using train_test_split():

In [56]:
```
## Test-Train Split
# def train_test(X,y):
#     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25,
#     # X_train #A matrix containing the predictors associated with the training
#     # X_test #A matrix containing the predictors associated with the test data
#     # y_train #A vector containing the class labels for the training observati
#     return X_train, X_test, y_train, y_test
# X_train, X_test, y_train, y_test = train_test(X,y)
```

Now I will use KFold. Here are two of the main benefits:

1. Introducing cross-validation into the process helps reduce the need for the validation set because we are now able to train and test on the same data.
   - https://towardsdatascience.com/complete-guide-to-pythons-cross-validation-with-examples-a9676b5cac12
2. Even though sklearn's train_test_split method is using a stratified split, which means that the train and test set have the same distribution as the target variable, it's still entirely possible to accidentally train on a subset that doesn't reflect the whole dataset. KFolds mitigates that threat, reducing the potential bias.

KFold function for Part 2 that replaces the need for using train_test_split:

In [102…
```
def kfold_train_test_split(X,y):

    #n_splits=10 means that the KFold will shift the test set 10 times
    #shuffle is False by default but if shuffle=True then splitting will be rand
    kf = KFold(n_splits=10, shuffle=True, random_state=313)
    kf.get_n_splits(X)
    # print(kf)

    # for train_index, test_index in kf.split(X,y):
    #      # print("TRAIN:", np.take(X,train_index), "TEST:", test_index)
    #      X_train, X_test = X[train_index], X[test_index]
    #      y_train, y_test = y[train_index], y[test_index]

         # getting this error:
         # KeyError: "None of [Int64Index([    0,     1,     2,     3,     5,
         # realize that I cannot use a pandas dataframe since KFold uses numpy a
         # so need to convert the pandas dataframes i'm using into numpy arrays
         # in order for this to work

    # convert pandas dataframe into numpy array:
    X_np = X.to_numpy()
    y_np = y.to_numpy()

    # split the new X_np and y_np into train and test
    for train_index, test_index in kf.split(X_np):
        # print("TRAIN:", np.take(X_np,train_index), "TEST:", np.take(X_np, test
        X_train, X_test = X_np[train_index], X_np[test_index]
        y_train, y_test = y_np[train_index], y_np[test_index]

        # now save the variables back as a pandas dataframe
        # X_train, X_test = pd.DataFrame(data=X_np[train_index], columns=['Red',
        # y_train, y_test = pd.DataFrame(data=y_np[train_index], columns=['Blue


    return X_train, X_test, y_train, y_test

X_train, X_test, y_train, y_test = kfold_train_test_split(X,y)
```

Data Leakage: Scaling our data

Now that we have X and y split into train and test sets, we can use those splits to scale our data to prevent data leakage.

In [140…
```
def scale_X_data(X_train_foo, X_test_foo):
    #then we will scale the data using StandardScaler
    scaler = preprocessing.StandardScaler().fit(X_train_foo)
```

```
    X_train_foo_scaled = scaler.transform(X_train_foo)
    X_test_foo_scaled = scaler.transform(X_test_foo)

    return X_train_foo_scaled, X_test_foo_scaled

X_train_scaled, X_test_scaled = scale_X_data(X_train, X_test)
```

Methods from Part 1

This next section is going to be applying the scaled data to the methods from Part 1 of the project.

Finding the optimal value of K for KNN

In [109…

```
#KNN-specific
def optimal_K(X_train_knn, X_test_knn, y_train_knn, y_test_knn):
    ### Testing for best/optimal K value
    # def find_optimal_k
    accuracy_rate_dict = {}
    for i in range(1,40):
        knn = neighbors.KNeighborsClassifier(n_neighbors=i)
        knn_pred = knn.fit(X_train_knn, y_train_knn).predict(X_test_knn)
        accuracy_rate_dict[accuracy_score(y_test_knn, knn_pred)] = i
    max(accuracy_rate_dict)
    best_k = accuracy_rate_dict[max(accuracy_rate_dict)] #20
    print('KNN performs best when the classifier k={}'.format(best_k))

    K = best_k #A value for K, the number of nearest neighbors to be used by the
    return K
```

KNN

This knn() function works rather differently from the other model-fitting functions that we have encountered thus far. Rather than a two-step approach in which we first fit the model and then we use the model to make predictions, knn() forms predictions using a single command. The function requires four inputs: X_train, X_test, y_train, and y_test.

In [290…

```
def knn(X_train_knn, X_test_knn, y_train_knn, y_test_knn):

    K = optimal_K(X_train_knn, X_test_knn, y_train_knn, y_test_knn)

    """Now the neighbors.KNeighborsClassifier() function can be used to predict
    classification of Blue Tarp, Rooftop, Soil, Various Non-Tarp, or Vegetation"
    # fit a model
    knn = neighbors.KNeighborsClassifier(n_neighbors = K)

    #predict classification
    knn_pred = knn.fit(X_train_knn, y_train_knn).predict(X_test_knn) #fit the mo

    # predict probabilities
    knn_probs = knn.fit(X_train_knn, y_train_knn).predict_proba(X_test_knn) #fit

    # keep probabilities for the positive outcome only
    knn_probs = knn_probs[:, 1]

    return knn_pred, knn_probs
```

In [271…
```
#unscaled data
knn_pred, knn_probs = knn(X_train, X_test, y_train, y_test)
```

```
KNN performs best when the classifier k=6
```

In [272…
```
#scaled data
knn_pred_scaled, knn_probs_scaled = knn(X_train_scaled, X_test_scaled, y_train,
```

```
KNN performs best when the classifier k=8
```

Our knn() and optimal_k() functions report that the optimal classifier is k=6 with unscaled data and k=8 when using the scaled data.

LDA -- Linear Discriminant Analysis

In [294…
```
def lda(X_train_lda, X_test_lda, y_train_lda, y_test_lda):
    # fit a model
    lda = LinearDiscriminantAnalysis()

    #predict classification
    lda_pred = lda.fit(X_train_lda, y_train_lda).predict(X_test_lda)

    # predict probabilities
    lda_probs = lda.fit(X_train_lda, y_train_lda).predict_proba(X_test_lda)

    # keep probabilities for the positive outcome only
    lda_probs = lda_probs[:, 1]

    return lda_pred, lda_probs

lda_pred_scaled, lda_probs_scaled = lda(X_train_scaled, X_test_scaled, y_train,
```

QDA -- Quadratic Discriminant Analysis

In [142…
```
def qda(X_train_qda, X_test_qda, y_train_qda, y_test_qda):
    # fit a model
    qda = QuadraticDiscriminantAnalysis()

    #predict classification
    qda_pred = qda.fit(X_train_qda, y_train_qda).predict(X_test_qda)

    # predict probabilities
    qda_probs = qda.fit(X_train_qda, y_train_qda).predict_proba(X_test_qda)

    # keep probabilities for the positive outcome only
    qda_probs = qda_probs[:, 1]

    return qda_pred, qda_probs
```

In [250…
```
#unscaled data
qda_pred, qda_probs = qda(X_train, X_test, y_train, y_test)
```

In [251…
```
#scaled data
qda_pred_scaled, qda_probs_scaled = qda(X_train_scaled, X_test_scaled, y_train,
```

Logistic Regression

In [143…
```
def logistic_regression(X_train_logreg, X_test_logreg, y_train_logreg, y_test_lo
    # fit a model
    model = LogisticRegression(solver='lbfgs')
```

```
    #predict classification
    logreg_pred = model.fit(X_train_logreg, y_train_logreg).predict(X_test_logre

    # predict probabilities
    logreg_probs = model.fit(X_train_logreg, y_train_logreg).predict_proba(X_tes

    # keep probabilities for the positive outcome only
    logreg_probs = logreg_probs[:, 1]

    return logreg_pred, logreg_probs
```

In [252…
```
#unscaled data
logreg_pred, logreg_probs = logistic_regression(X_train, X_test, y_train, y_test
```

In [253…
```
#scaled data
logreg_pred_scaled, logreg_probs_scaled = logistic_regression(X_train_scaled, X_
```

Part 2

Now we will build out the Random Forest and Support Vector Machine models.

A note on decision trees: individual decision trees are fragile in the sense that they are only as good as the data on which they are trained. If the underlying data changes, then so will the decision tree's prediction.

Random Forests reduce this variance by creating decision trees for many subsets of the data and then taking the average of all those decision trees. In a typical Random Forest, a random subset of the features are chosen for each decision tree. In a bagging model, all of the features are chosen for each decision tree.

Bagging

In [144…
```
def bagging(X_train_bag, X_test_bag, y_train_bag, y_test_bag):
    # fit a model
    rfc = RandomForestClassifier(max_features=X_train.shape[1],random_state=313)

    #predict classification
    bagging_pred = rfc.fit(X_train_bag, y_train_bag).predict(X_test_bag)

    # predict probabilities
    bagging_probs = rfc.fit(X_train_bag, y_train_bag).predict_proba(X_test_bag)

    # keep probabilities for the positive outcome only
    bagging_probs = bagging_probs[:, 1]

    return bagging_pred, bagging_probs, rfc
```

In [254…
```
#unscaled data
bagging_pred, bagging_probs, rfc = bagging(X_train, X_test, y_train, y_test)
```

In [255…
```
#scaled data
bagging_pred_scaled, bagging_probs_scaled, rfc_scaled = bagging(X_train_scaled,
```

Now let's evaluate how the model performed:

In [145…
```
# Evaluating Performance:
def eval_perform_rf(rfc, X_whole_dataset, y_whole_dataset, y_test_rf, bagging_pr
    # calculate the cross_val_score for the bagging model, using rfc from the ba
    bagging_cv_score = cross_val_score(rfc, X_whole_dataset, y_whole_dataset, cv
    # using n_jobs=-1 to speed up the run time

    for i in range(len(bagging_cv_score)):
        print('CV={}: {}'.format(i+1,bagging_cv_score[i]))
    print('\n')

    print("=== Confusion Matrix ===")
    print(confusion_matrix(y_test_rf, bagging_pred))
    print('\n')

    print("=== Classification Report ===")
    print(classification_report(y_test_rf, bagging_pred))
    print('\n')

    print("=== All AUC Scores ===")
    print(bagging_cv_score)
    print('\n')

    print("=== Mean AUC Score ===")
    print("Mean AUC Score - Random Forest: ", bagging_cv_score.mean())
```

In [256…
```
#unscaled data
eval_perform_rf(rfc, X, y, y_test, bagging_pred)
```

```
CV=1: 1.0
CV=2: 0.9999482470298647
CV=3: 0.9975247524752475
CV=4: 1.0
CV=5: 1.0
CV=6: 1.0
CV=7: 0.9868721313490382
CV=8: 0.9940132325875515
CV=9: 0.9081534378527693
CV=10: 0.9993541574954349


=== Confusion Matrix ===
[[6123    4]
 [   7  190]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      6127
           1       0.98      0.96      0.97       197

    accuracy                           1.00      6324
   macro avg       0.99      0.98      0.99      6324
weighted avg       1.00      1.00      1.00      6324



=== All AUC Scores ===
[1.         0.99994825 0.99752475 1.         1.         1.
 0.98687213 0.99401323 0.90815344 0.99935416]
```

```
=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9885865958789907
```

In [257…

```
#scaled data
eval_perform_rf(rfc_scaled, X, y, y_test, bagging_pred_scaled)
```

```
CV=1: 1.0
CV=2: 0.9999482470298647
CV=3: 0.9975247524752475
CV=4: 1.0
CV=5: 1.0
CV=6: 1.0
CV=7: 0.9868721313490382
CV=8: 0.9940132325875515
CV=9: 0.9081534378527693
CV=10: 0.9993541574954349


=== Confusion Matrix ===
[[6123    4]
 [   7  190]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      6127
           1       0.98      0.96      0.97       197

    accuracy                           1.00      6324
   macro avg       0.99      0.98      0.99      6324
weighted avg       1.00      1.00      1.00      6324



=== All AUC Scores ===
[1.         0.99994825 0.99752475 1.         1.         1.
 0.98687213 0.99401323 0.90815344 0.99935416]


=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9885865958789907
```

This is pretty great. 98.86% accuracy for the AUC score.

- The confusion matrix is useful for giving you false positives and false negatives.
- The classification report tells you the accuracy of your model.
- The ROC curve plots out the true positive rate versus the false positive rate at various thresholds.
- The roc_auc scoring used in the cross-validation model shows the area under the ROC curve.

We'll evaluate our model's score based on the roc_auc score (stored in bagging_cv_score.mean() and returned as "Mean AUC Score - Random Forest"), which is 0.9886 in this model.

The next thing we should do is tune our hyperparameters to see if we can improve the performance of the model. https://medium.com/@hjhuney/implementing-a-random-forest-classification-model-in-python-583891c99652

https://towardsdatascience.com/hyperparameter-tuning-the-random-forest-in-python-using-scikit-learn-28d2aa77dd74

Tuning Hyperparameters

William Koehrsen's article, "Hyperparameter Tuning the Random Forest in Python", recommends using a full grid of hyperparameters. I actually attempted to do something similar to what he laid out in his article (see final.py for my attempt) but this took way too much time, so I ended up focusing on 3 hyperparameters.

Initially I took the lead of Jason Brownlee (Medium article above) and used n_estimators, max_features, and max_depth, but I end up revising that later on in the SVM section when I'm using the entire holdout dataset. I'll provide more explanation later on as to why I valued some hyperparameters over others.

In [146…

```
# Tuning Hyperparameters for the Random Forest model

def rf_tune_hyperparameters(rfc, X_train_rf_tune, y_train_rf_tune):
    #n_estimators determines the number of trees in the random forest
    #take 11 values of n_estimators starting from 100 and ending with 2000, equa

    # n_estimators = [int(x) for x in np.linspace(start = 100, stop = 2000, num
    n_estimators = [10, 50, 100, 500, 1000]

    # number of features at every split
    max_features = ['auto', 'sqrt','log2', np.random.randint(1,4)] #default is a

    # max_depth determines the maximum depth of the tree
    #take 11 values of max_depth starting from 100 and ending with 500, equally
    max_depth = [int(x) for x in np.linspace(100, 500, num = 11)]
    # max_depth = [2,3,4,5,6,7,8,9, 10, None] #None is the default for max_depth
    # max_depth = [2,3,4]

    # criterion = ['gini','entropy']

    # create grid of these parameters that will be passed into the searchCV func
    param_grid = {
     'n_estimators': n_estimators,
     'max_features': max_features,
     'max_depth': max_depth
     }

    # GridSearchCV of the parameters
    rfc_gridsearch = GridSearchCV(
        rfc,
        param_grid,
        # verbose=2,
        n_jobs=-1
        )

    #I was getting some errors using pandas dataframes with this model fit
    #so changing to numpy array
    #rfc_gridsearch.fit(X_train_rf.to_numpy(), y_train_rf.to_numpy())
    rfc_gridsearch.fit(X_train_rf_tune, y_train_rf_tune)
    # this is a pretty data intensive model fit which is why i have set n_jobs t
    #in the RandomizedSearchCV(), which means that all of my processors are bein
    #used in parallel to run this fit
```

```
    # print the results:
    # print(rfc_random.best_params_)

    #now store the optimal parameters and then return them for use in other func
    optimal_n_estimators = rfc_gridsearch.best_params_['n_estimators']
    optimal_max_features = rfc_gridsearch.best_params_['max_features']
    optimal_max_depth = rfc_gridsearch.best_params_['max_depth']
    # optimal_criterion = rfc_gridsearch.best_params_['criterion']

    print("=== Optimal n_estimators ===")
    print(optimal_n_estimators)
    print('\n')

    print("=== Optimal max_features ===")
    print(optimal_max_features)
    print('\n')

    print("=== Optimal max_depth ===")
    print(optimal_max_depth)
    print('\n')

    return rfc_gridsearch, optimal_n_estimators, optimal_max_features, optimal_m
```

In [259…

```
# scaled data
# comment this next line out if you don't want to wait for the whole thing to ru
rfc_gridsearch_scaled, optimal_n_estimators_scaled, optimal_max_features_scaled,
```

```
=== Optimal n_estimators ===
500


=== Optimal max_features ===
auto


=== Optimal max_depth ===
100
```

Now we can plug these back values into the model to see if it improves our model's performance.

In [260…

```
def rf_optimal(optimal_n, optimal_feat, optimal_depth):
    rfc_optimal = RandomForestClassifier(n_estimators=optimal_n_estimators, max_
    rfc_optimal.fit(X_train,y_train)
    rfc_optimal_predict = rfc_optimal.predict(X_test)
    rfc_optimal_cv_score = cross_val_score(rfc_optimal, X, y, cv=10, scoring='ro
    for i in range(len(rfc_optimal_cv_score)):
        print('CV={}: {}'.format(i+1,rfc_optimal_cv_score[i]))
    print('\n')

    print("=== Confusion Matrix ===")
    print(confusion_matrix(y_test, rfc_optimal_predict))
    print('\n')
    # === Confusion Matrix ===
    # [[15289    18]
    #  [   25   479]]

    print("=== Classification Report ===")
```

```
    print(classification_report(y_test, rfc_optimal_predict))
    print('\n')

    print("=== All AUC Scores ===")
    print(rfc_optimal_cv_score)
    print('\n')

    print("=== Mean AUC Score ===")
    print("Mean AUC Score - Random Forest: ", rfc_optimal_cv_score.mean())

    return rfc_optimal_cv_score
```

In [261…

```
#unscaled data
rf_opimal_cv_score = rf_optimal(optimal_n_estimators, optimal_max_features, opti
```

```
CV=1: 1.0
CV=2: 0.9865676783294142
CV=3: 0.9998859817376706
CV=4: 1.0
CV=5: 1.0
CV=6: 1.0
CV=7: 0.9988040212057796
CV=8: 0.998960493076423
CV=9: 0.9393924201306115
CV=10: 0.9994366482826222


=== Confusion Matrix ===
[[6125    2]
 [   7  190]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      6127
           1       0.99      0.96      0.98       197

    accuracy                           1.00      6324
   macro avg       0.99      0.98      0.99      6324
weighted avg       1.00      1.00      1.00      6324



=== All AUC Scores ===
[1.         0.98656768 0.99988598 1.         1.         1.
 0.99880402 0.99896049 0.93939242 0.99943665]


=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9923047242762519
```

In [262…

```
#scaled data
rf_opimal_cv_score_scaled = rf_optimal(optimal_n_estimators_scaled, optimal_max_
```

```
CV=1: 1.0
CV=2: 0.9865676783294142
CV=3: 0.9998859817376706
CV=4: 1.0
CV=5: 1.0
CV=6: 1.0
CV=7: 0.9988040212057796
CV=8: 0.998960493076423
CV=9: 0.9393924201306115
```

```
CV=10: 0.9994366482826222


=== Confusion Matrix ===
[[6125    2]
 [   7  190]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       1.00      1.00      1.00      6127
           1       0.99      0.96      0.98       197

    accuracy                           1.00      6324
   macro avg       0.99      0.98      0.99      6324
weighted avg       1.00      1.00      1.00      6324



=== All AUC Scores ===
[1.         0.98656768 0.99988598 1.         1.         1.
 0.99880402 0.99896049 0.93939242 0.99943665]


=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9923047242762519
```

This also performs incredibly well: 99.23% accuracy - AUC score. This is slightly higher than bagging but the differences seen in this confusion matrix can easily be explained by variance. There is inherent uncertainty when selecting optimal parameters for our models. This is undoubetably a great fit but the differences between this "optimal" model and the bagging model are too small to confidently say which is better.

Support Vector Machines

Now we can create the SVM model for filling out the table

In [263…

```
### SVM

def svm(X_train_svm, X_test_svm, y_train_svm, y_test_svm):
    # fit a model
    clf_svm = SVC(random_state=313, probability=True)
    clf_svm.fit(X_train_svm, y_train_svm)

    #predict classification
    clf_svm_pred = clf_svm.fit(X_train_svm, y_train_svm).predict(X_test_svm)

    # predict probabilities
    clf_svm_probs = clf_svm.fit(X_train_svm, y_train_svm).predict_proba(X_test_s
    #AttributeError: predict_proba is not available when  probability=False

    # keep probabilities for the positive outcome only
    clf_svm_probs = clf_svm_probs[:, 1]

    return clf_svm_pred, clf_svm_probs, clf_svm
```

In [264…

```
clf_svm_pred, clf_svm_probs, clf_svm = svm(X_train_scaled, X_test_scaled, y_trai
```

Downsampling

The 63,241 observations in the pixel data will take a long time for the support vector machine to run, so downsampling will make this process quicker. Support Vector Machines are great with small datasets, but they aren't so great with large ones. The pixels dataset is large enough to take a long time to optimize with a typical cross validation approach, so taking a subset of this dataset and feeding it to the SVM model will give us a good idea of how the model behaves with our data, without needing to run through every observation.

For this project I downsample both categories, Blue Tarp and Non Blue Tarp, to 2,022 each. There are only 2,022 observations where y = Blue Tarp, so I wanted to include as many as possible to make sure the model is not suffering from this reduction in training size.

In [124…

```
def downsample_data(num_samples):
    #To ensure that i actually get 2022 samples for each category, I'll start by
    #splitting the data into two dataframes, one for Class=1 (blue tarp) and one
    #for Class=0 (not blue tarp)
    pixels_blue_tarp = pixels[y==1] #pixels[pixels['Class']==1]
    pixels_not_blue_tarp = pixels[y==0] # pixels[pixels['Class']==0]

    #let's see how this breaks down
    pixels_blue_tarp.shape #(2022, 4) ... 2022 observations
    pixels_not_blue_tarp.shape #(61219, 4) ... 61219 observations
    sum(y)/len(y) #0.03197292895431761
    #This is definitely an unbalanced dataset where there are way more examples
    #blue tarp observations than there are blue tarp observations (this is expec

    #now downsample the pixels_blue_tarp dataframe
    pixels_blue_tarp_downsampled = resample(pixels_blue_tarp,
                                            replace=False,
                                            n_samples=num_samples, #pass in num_
                                            random_state=313)

    #now downsample the pixels_not_blue_tarp dataframe
    pixels_not_blue_tarp_downsampled = resample(pixels_not_blue_tarp,
                                            replace=False,
                                            n_samples=num_samples, #pass in num_
                                            random_state=313)

    #now merge the two downsampled dataframes back into a single dataframe
    pixels_downsampled = pd.concat([pixels_blue_tarp_downsampled, pixels_not_blu
    #then print out the total number of samples (should be 2000 or 2*num_samples
    len(pixels_downsampled) #4044

    return pixels_downsampled

pixels_downsampled = downsample_data(num_samples=2022)
```

In [129…

```
pixels_downsampled.head()
```

Out[129…

	Class	Red	Green	Blue
63068	1	125	116	115
61804	1	195	224	255
62869	1	173	195	206

	Class	Red	Green	Blue
61303	1	171	193	225
62225	1	144	163	216

In [130…
```
pixels_downsampled.shape
```

Out[130… (4044, 4)

Building the SVM

Now that we have formatted our downsampled data, we can start building the support vector machine. First, we split the data into two parts:

1. the column of data we will use to make classifications
2. the column of data that we want to predict

In [131…
```
#split this data using the load_data() from above and pass it the downsampled da
X_downsampled, y_downsampled = load_data(pixels_downsampled)
```

In [132…
```
X_downsampled, y_downsampled
```

Out[132…
```
(       Red  Green  Blue
 63068  125    116   115
 61804  195    224   255
 62869  173    195   206
 61303  171    193   225
 62225  144    163   216
 ...    ...    ...   ...
 27592  255    252   187
 27634  227    165   150
 23947   90     84    64
 57879  168    143   108
 27032  255    230   173

 [4044 rows x 3 columns],
 63068    1
 61804    1
 62869    1
 61303    1
 62225    1
         ..
 27592    0
 27634    0
 23947    0
 57879    0
 27032    0
 Name: Class, Length: 4044, dtype: int64)
```

Nothing to one hot encode in X. Everything looks good.

But the radial basis function that we use with our support vector machine assumes that the data we feed it are centered and scaled, so we need to make sure that each column should have a mean value=0 and a standard deviation=1 for both the training and testing dataset.

We'll use the kfold_train_test_split() function we wrote earlier to split the data into training data and testing data.

In [136…
```
X_train_downsampled, X_test_downsampled, y_train_downsampled, y_test_downsampled
```

Then we will scale the data using the scale_X_data() from before.

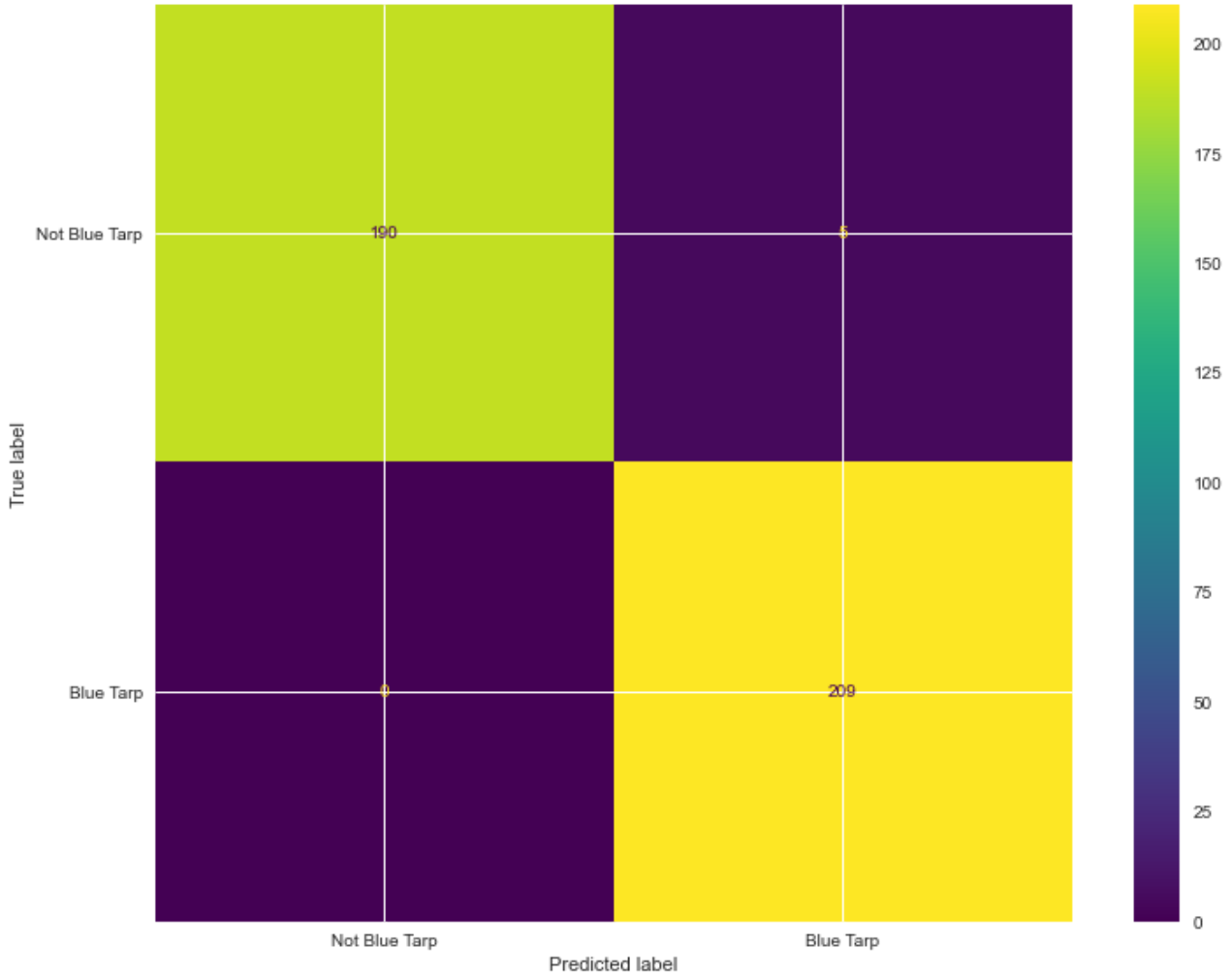
In [147…
```
X_train_downsampled_scaled, X_test_downsampled_scaled = scale_X_data(X_train_dow
```

In [150…
```
X_train_downsampled_scaled, X_test_downsampled_scaled
```

Out[150…
```
array([[-0.70984281, -0.89687171, -0.74529512],
       [ 0.43087686,  0.85610623,  1.33746869],
       [ 0.07236496,  0.38539919,  0.60850136],
       ...,
       [-1.28020265, -1.41627258, -1.50401623],
       [-0.00911501, -0.45862722, -0.84943332],
       [ 1.40863658,  0.95349389,  0.11756417]])
```

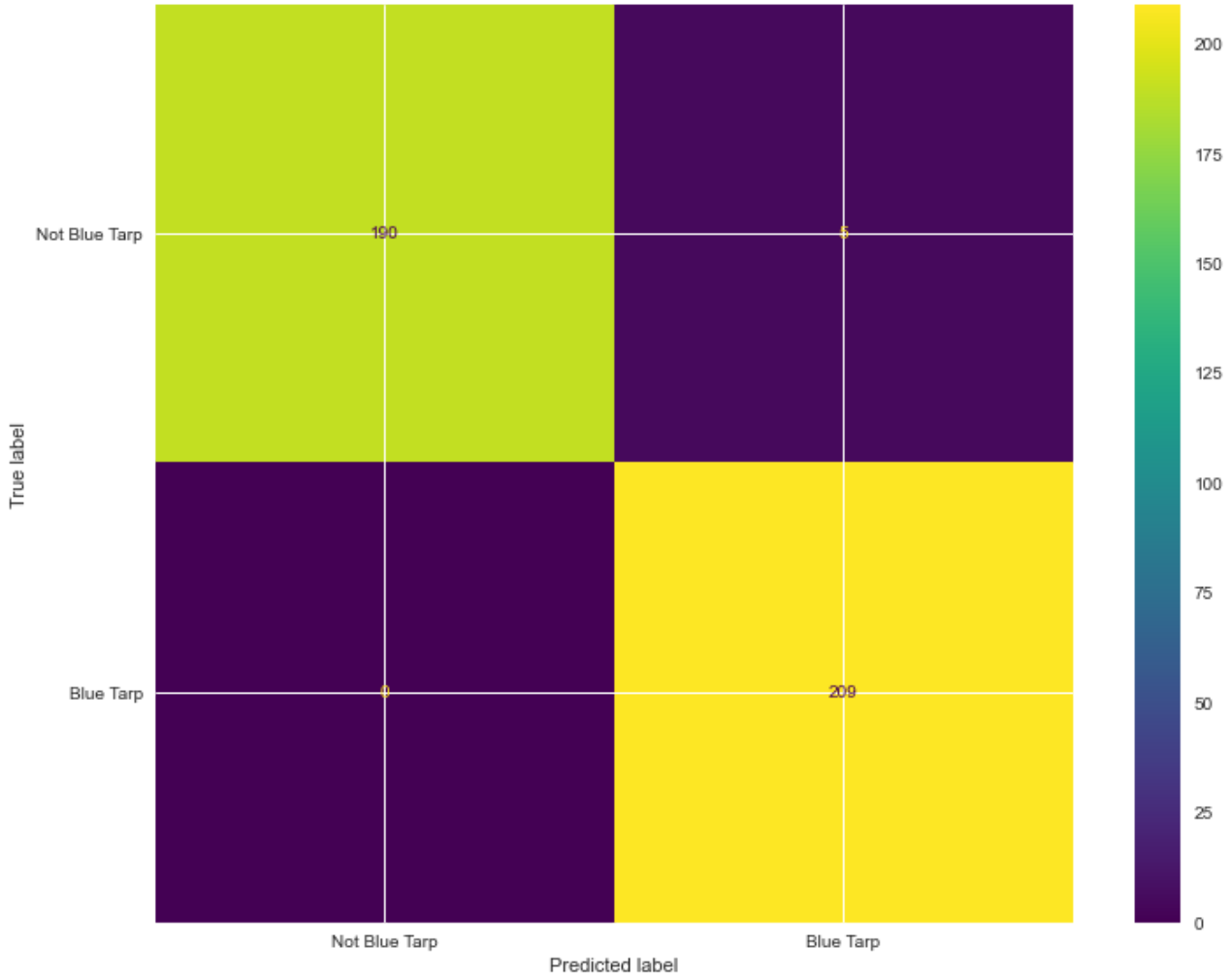
In [374…
```
# This function builds the SVM model and can take in either downsampled data or
# but because of SVM's structure, it expects scaled data
def build_svm(X_train_foo_scaled, X_test_foo_scaled, y_train_foo, y_test_foo, cl
    # Now that we have our data split into test and train we can build our preli
    # support vector machine

    # clf_svm = SVC(random_state=313)
    clf_svm.fit(X_train_foo_scaled, y_train_foo)

    #that's it, that's our support vector machine for classification
    #now we can draw a confusion matrix and see how it performs on the test data

    plot_confusion_matrix(clf_svm,
                          X_test_foo_scaled,
                          y_test_foo,
                          values_format='d',
                          display_labels=["Not Blue Tarp", "Blue Tarp"])
```

In [375…
```
#unscaled data
build_svm(X_train_downsampled, X_test_downsampled, y_train_downsampled, y_test_d
```



In [179…

```
#scaled data
build_svm(X_train_downsampled_scaled,X_test_downsampled_scaled,y_train_downsampl
```



This performs incredibly well, which always makes me pause and think whether I might be overfitting. Are these tests really performing so well or is this just evidence of random forest and SVMs being better solutions for this kind of classification problem than the methods used in Part 1?

I will use cross validation to optimize the parameters to see if I can improve predictions at all but honestly, these numbers are already fantastic. My next step would be to run the entire dataset (without downsampling) to see how it performs.

In [238…
```
### SVM -- now we can actually build the SVM model
# use the svm() function I defined above
clf_svm_pred_downsampled, clf_svm_probs_downsampled, clf_svm_downsampled = svm(X
```

When we are optimizing a support vector machine we are attempting to find the best value for gamma and the regularization parameter C that will improve the prediction / classification accuracy of the model. Since we are looking to optimize two parameters (gamma and C) we will use GridSearchCV(), specifying several values for gamma and C and then letting the function determine the optimal combination (that's the great thing about using GridSerachCV() for problems like this: it tests all possible combinations of parameters for us and all we have to do is plug in some options).

The default values for the SVC parameters in sklearn.svm are:

- C=1.0
- kernel='rbf
- degree=3
- gamma='scale' (which is 1/(n_features*X.var()) while 'auto' uses 1/n_features

GridSearchSV in sklearn.model_selection has a parameter called param_grid where you can pass in a dictionary of parameters. I will pass in all of the default SVC parameters and then some extra parameters so that the GridSearchCV() can find the optimal combination from many options.

My assumption is that 'rbf' will work the best for the pixel data, so I might not need to have all these options for kernel, but I am including 'linear','poly', and 'sigmoid' anyway.

In [154…
```
def optimizing_svm():
    param_grid = [
      {'C': np.arange(0.001,5.001,.1),
       'degree':[1,2,3,4],
       'gamma': ['scale', 'auto', 1, 0.1, 0.01, 0.001, 0.0001],
       'kernel': ['rbf','linear','poly','sigmoid']},
    ]

    optimal_params = GridSearchCV(
        SVC(),
        param_grid,
        cv=5,
        scoring='accuracy',
        n_jobs=-1)

    return optimal_params
```

In [155…
```
optimal_params = optimizing_svm()
print(optimal_params)
```

Now that we have the optimal parameters for the SVM model, we need to fit the model with the optimal parameter grid. This takes a little while but here we start searching through for the best options...

In [157…
```
optimal_params.fit(X_train_downsampled_scaled, y_train_downsampled)
```

Out[157…
```
GridSearchCV(cv=5, estimator=SVC(), n_jobs=-1,
             param_grid=[{'C': array([1.000e-03, 1.010e-01, 2.010e-01, 3.010e-0
1, 4.010e-01, 5.010e-01,
       6.010e-01, 7.010e-01, 8.010e-01, 9.010e-01, 1.001e+00, 1.101e+00,
       1.201e+00, 1.301e+00, 1.401e+00, 1.501e+00, 1.601e+00, 1.701e+00,
       1.801e+00, 1.901e+00, 2.001e+00, 2.101e+00, 2.201e+00, 2.301e+00,
       2.401e+00, 2.501e+00, 2.601e+00, 2.701e+00, 2.801e+00, 2.901e+00,
       3.001e+00, 3.101e+00, 3.201e+00, 3.301e+00, 3.401e+00, 3.501e+00,
       3.601e+00, 3.701e+00, 3.801e+00, 3.901e+00, 4.001e+00, 4.101e+00,
       4.201e+00, 4.301e+00, 4.401e+00, 4.501e+00, 4.601e+00, 4.701e+00,
       4.801e+00, 4.901e+00]),
                         'degree': [1, 2, 3, 4],
                         'gamma': ['scale', 'auto', 1, 0.1, 0.01, 0.001,
                                   0.0001],
                         'kernel': ['rbf', 'linear', 'poly', 'sigmoid']}],
             scoring='accuracy')
```

In [161…
```
# Save the optimal parameters down in their own varaibles
```

```
best_C = optimal_params.best_params_['C']
best_degree = optimal_params.best_params_['degree']
best_gamma = optimal_params.best_params_['gamma']
best_kernel = optimal_params.best_params_['kernel']

# Then print out the optimal parameters for the model
print('''The best cost for clf_svm is: {}
the best degree for clf_svm is: {}
the best gamma for clf_svm is: {}
the best kernel for clf_svm is: '{}'
      '''.format(
      best_C,
      best_degree,
      best_gamma,
      best_kernel
))
```

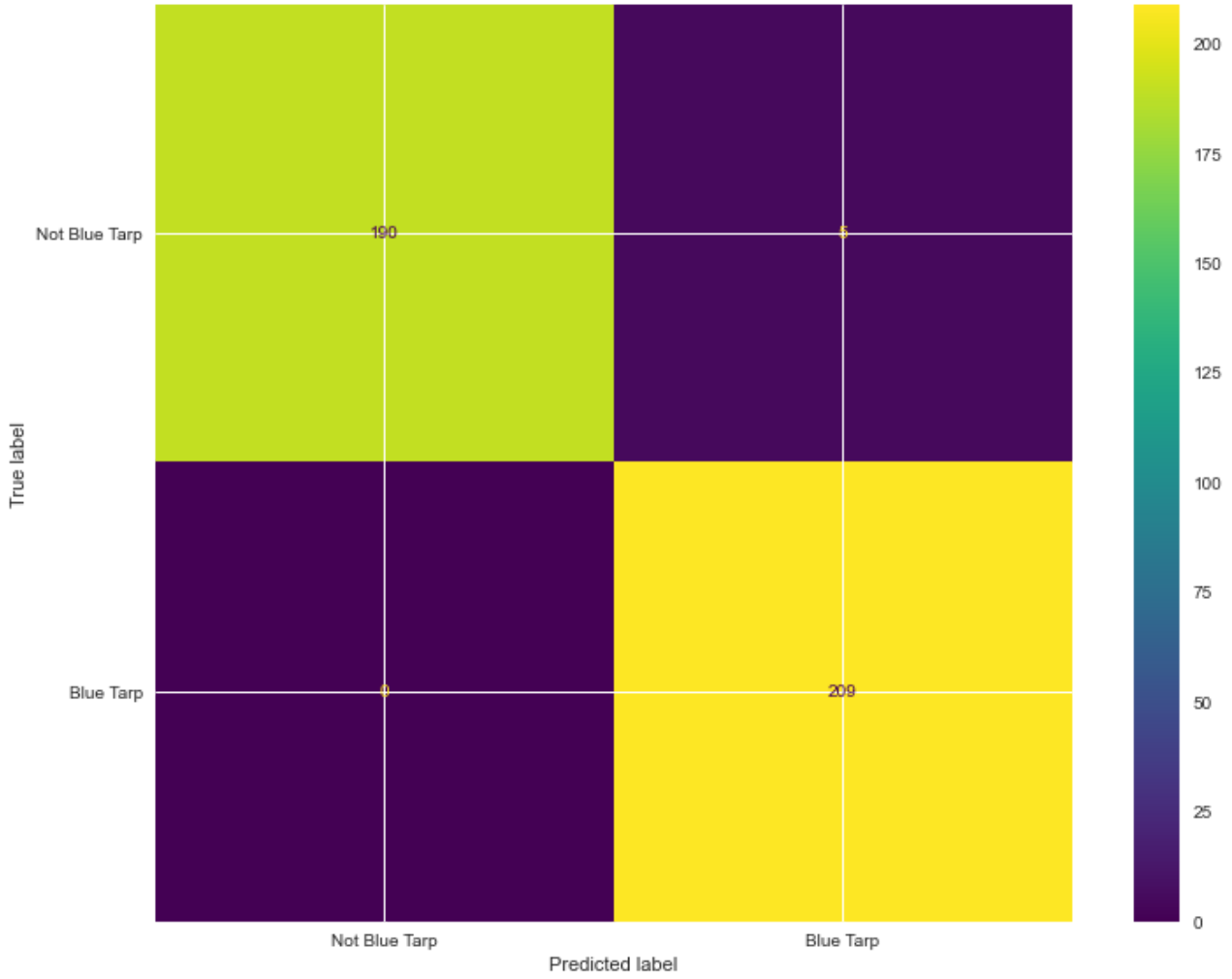
```
The best cost for clf_svm is: 3.801
the best degree for clf_svm is: 1
the best gamma for clf_svm is: 1
the best kernel for clf_svm is: 'rbf'
```

As expected, the best kernel for the SVM model is 'rbf'.

Now we can use these optimal parameters to fit a new model:

In [162…

```
clf_svm_optimal = SVC(random_state=313, C=best_C, degree=best_degree, gamma=best

build_svm(X_train_downsampled_scaled, X_test_downsampled_scaled, y_train_downsam
```



Interestingly, this is an identical confusion matrix to the one we built before.

Completing Table 2

Accuracy

In [206…

```
def accuracy(y_test, pred, method):
    accuracy = accuracy_score(y_test, pred)
    print('{} Accuracy: {}'.format(method,str(accuracy)))
    print('{} Test Error: {}'.format(method, str(1 - accuracy )))
    """that's actually not very helpful because this is scoring across classes,
    isn't giving us a good view of the accuracy of the model...use this instead:

    return accuracy
    # ^^ there is a severe imbalance
    #https://machinelearningmastery.com/threshold-moving-for-imbalanced-classifi
    #need to move threshold to accommodate this imbalance
    #"It has been stated that trying other methods, such as sampling, without tr
    #Pages 72, Imbalanced Learning: Foundations, Algorithms, and Applications, 2

knn_accuracy = accuracy(y_test,knn_pred,'KNN')
lda_accuracy = accuracy(y_test,lda_pred,'LDA')
qda_accuracy = accuracy(y_test,qda_pred,'QDA')
logreg_accuracy = accuracy(y_test,logreg_pred,'Logistic Regression')
bagging_accuracy = accuracy(y_test, bagging_pred,'Bagging')
svm_accuracy = accuracy(y_test_downsampled, clf_svm_pred, 'SVM')
```

```
KNN Accuracy: 0.9987349778621126
KNN Test Error: 0.0012650221378873727
LDA Accuracy: 0.9830803289057558
LDA Test Error: 0.016919671094244193
QDA Accuracy: 0.9955724225173941
QDA Test Error: 0.0044275774826059155
Logistic Regression Accuracy: 0.9966793168880456
Logistic Regression Test Error: 0.0033206831119544367
Bagging Accuracy: 0.9982605945604048
Bagging Test Error: 0.0017394054395951652
SVM Accuracy: 0.9876237623762376
SVM Test Error: 0.012376237623762387
```

AUC

We can then use the roc_auc_score() function to calculate the true-positive rate and false-positive rate for the predictions using a set of thresholds that can then be used to create a ROC Curve plot.

In [209…

```
def calculate_AUC(y_test, prob):
    # calculate scores
    auc = roc_auc_score(y_test, prob)
    return auc

auc_KNN = calculate_AUC(y_test, knn_pred)
print('auc_KNN:', auc_KNN)
auc_LDA = calculate_AUC(y_test, lda_pred)
print('auc_QDA:',auc_LDA)
auc_QDA = calculate_AUC(y_test, qda_pred)
print('auc_QDA:', auc_QDA)
auc_LogisticRegression = calculate_AUC(y_test, logreg_pred)
print('auc_LogisticRegression:',auc_LogisticRegression)
auc_Bagging = calculate_AUC(y_test, bagging_pred)
print('auc_Bagging:',auc_Bagging)
auc_SVM = calculate_AUC(y_test_downsampled, clf_svm_pred)
print('auc_SVM:',auc_SVM)
```

```
auc_KNN: 0.9870648266514446
auc_QDA: 0.8930095549448682
auc_QDA: 0.9363034053316476
auc_LogisticRegression: 0.9614392979729398
auc_Bagging: 0.9819070785132629
auc_SVM: 0.9871794871794871
```

Threshold for ROC

In [327…

```
def best_threshold(fpr, tpr, thresholds):
    # Youden's J-statistic for calculating optimal threshold
    # https://en.wikipedia.org/wiki/Youden%27s_J_statistic
    J = tpr - fpr

    best_index = 0
    #loop through J and set 'best_index' to whichever corresponds to the max(J)
    for index,value in enumerate(J):
        if value==max(J):
            best_index=index
            # print(best_index)

    best_thresh = thresholds[best_index]
    print('Best Threshold={:f}'.format(best_thresh))
    return best_index, best_thresh
```

ROC

In [328…
```
def calculate_ROC(y_test, prob, Type):
    # calculate roc curves
    fpr, tpr, thresholds = roc_curve(y_test, prob)

    # I cannot figure out why there are so fewer threshold values for KNN than f
    #print(thresholds)
    #fpr, tpr, threshold = roc_curve(y_test, knn_probs_scaled)
    #roc_auc = auc(fpr, tpr)
    #roc_auc

    best_index, best_thresh = best_threshold(fpr, tpr, thresholds)
    # plot the roc curve for the model
    plt.plot([0,1], [0,1], linestyle='--', label='No Skill')
    plt.plot(fpr, tpr, marker='.', label='{} ROC'.format(Type))
    #best threshold
    plt.scatter(fpr[best_index], tpr[best_index], marker='o', color='black', lab
    # axis labels
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend()
    # show the plot
    plt.show()
```

There's something strange going on with the knn_probs. I was having problems with my KNN graph in Part 1 also, and I still don't know what's going on.

In [329…
```
knn_probs, knn_probs_scaled, lda_probs, qda_probs, logreg_probs, bagging_probs,
```

Out[329…
```
(array([0., 0., 0., ..., 1., 1., 1.]),
 array([0., 0., 0., ..., 1., 1., 1.]),
 array([2.05501771e-07, 6.29110862e-08, 2.45905046e-08, ...,
        7.69431324e-01, 2.02634597e-01, 3.85522437e-01]),
 array([3.01152104e-05, 5.54408800e-05, 4.73949799e-05, ...,
        9.55472207e-01, 6.88937343e-01, 8.17629709e-01]),
 array([6.05578009e-04, 5.47287018e-05, 6.51550577e-06, ...,
        9.95684810e-01, 9.75742188e-01, 9.93962442e-01]),
 array([0.  , 0.  , 0.  , ..., 1.  , 0.99, 1.  ]),
 array([2.41113360e-04, 1.61771099e-04, 1.48139130e-04, ...,
        9.94769770e-01, 9.92174498e-01, 9.97303814e-01]),
 array([0, 0, 0, ..., 1, 1, 1]))
```

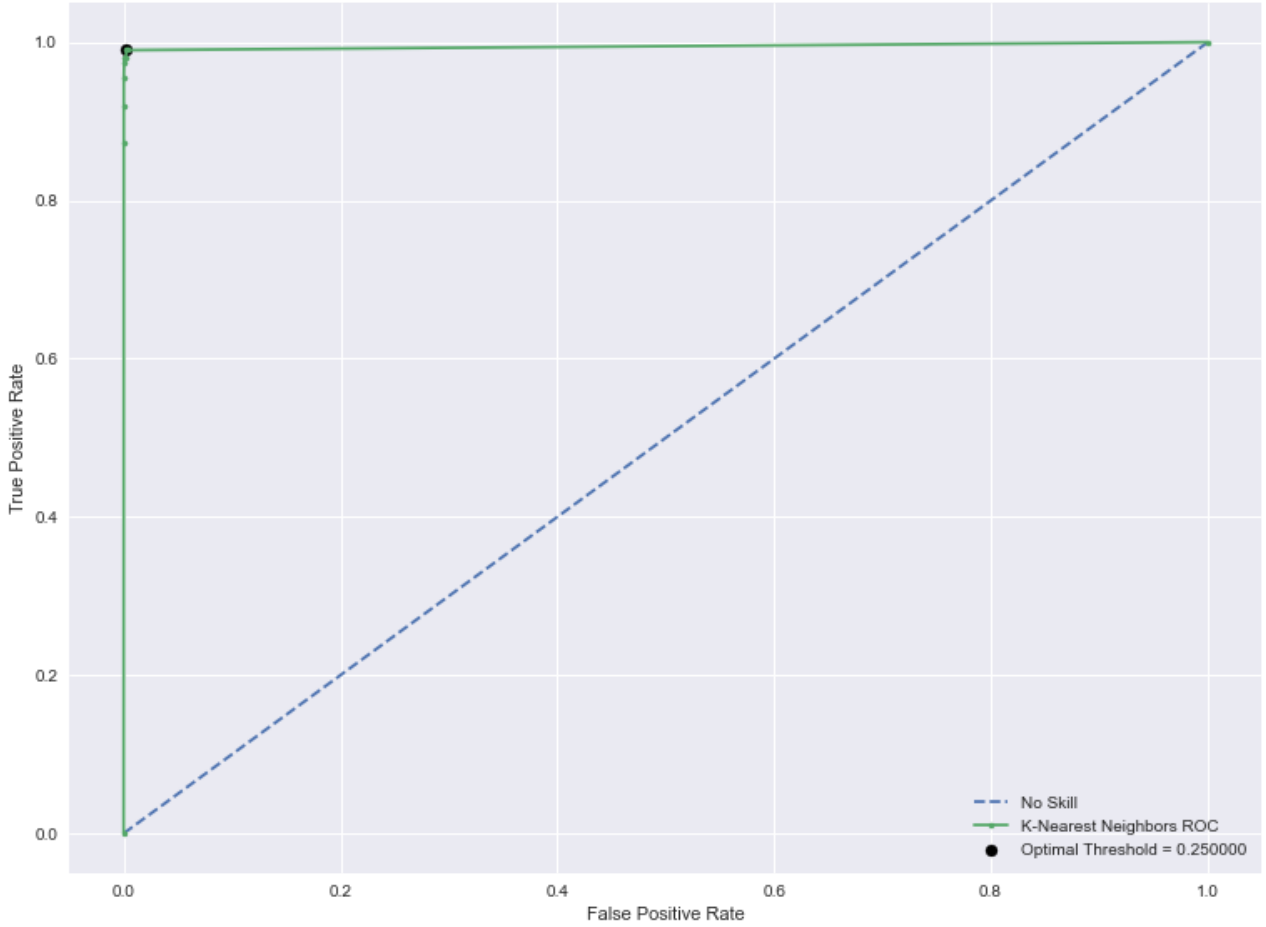
There are significantly fewer unique values in the KNN compared to all the others. However, I am unable to pinpoint why this is happening.

In [330…
```
np.unique(knn_probs_scaled), np.unique(bagging_probs), np.unique(knn_probs)
```

Out[330…
```
(array([0.   , 0.125, 0.25 , 0.375, 0.5  , 0.625, 0.75 , 0.875, 1.   ]),
 array([0.        , 0.01      , 0.01333333, 0.02      , 0.03      ,
        0.04      , 0.04866667, 0.06      , 0.07037852, 0.08      ,
        0.1       , 0.11      , 0.16      , 0.18      , 0.2       ,
        0.22      , 0.25      , 0.27      , 0.29      , 0.37      ,
        0.39      , 0.4       , 0.59      , 0.61      , 0.64      ,
        0.66      , 0.67      , 0.68      , 0.6975    , 0.75      ,
        0.78      , 0.8       , 0.83      , 0.87583333, 0.8795029 ,
        0.9       , 0.91      , 0.92      , 0.93      , 0.94      ,
        0.95      , 0.97      , 0.98      , 0.986     , 0.99      ,
        0.99166667, 1.        ]),
 array([0.        , 0.16666667, 0.33333333, 0.5       , 0.66666667,
        0.83333333, 1.        ]))
```

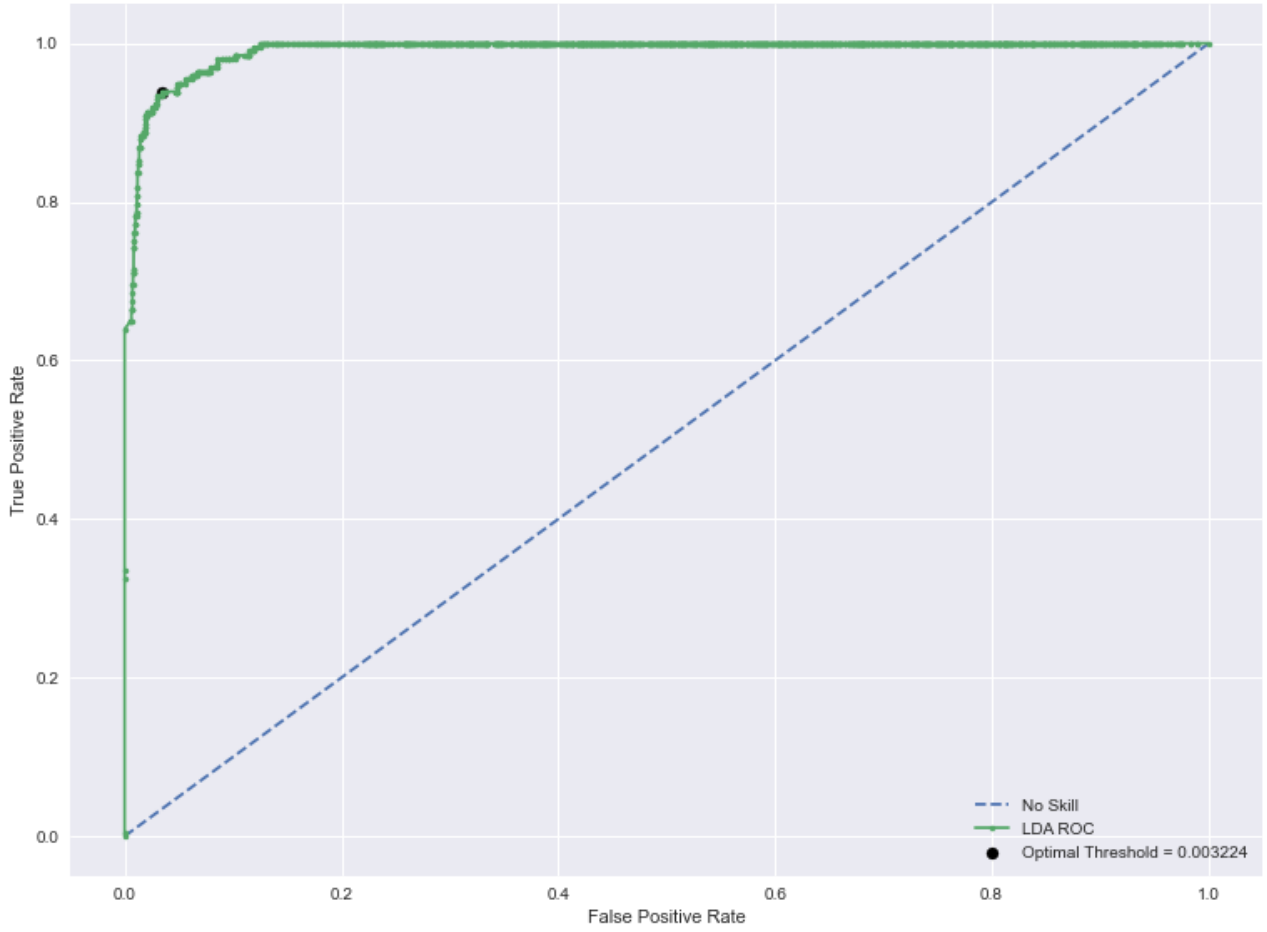
In [331…
```
roc_KNN = calculate_ROC(y_test, knn_probs_scaled, Type='K-Nearest Neighbors')
#roc_KNN = roc_curve(y_test, knn_probs_scaled)
#fpr, tpr, threshold = roc_curve(y_test, knn_probs_scaled)
#roc_auc = auc(fpr, tpr)
#roc_auc
```

```
Best Threshold=0.250000
```



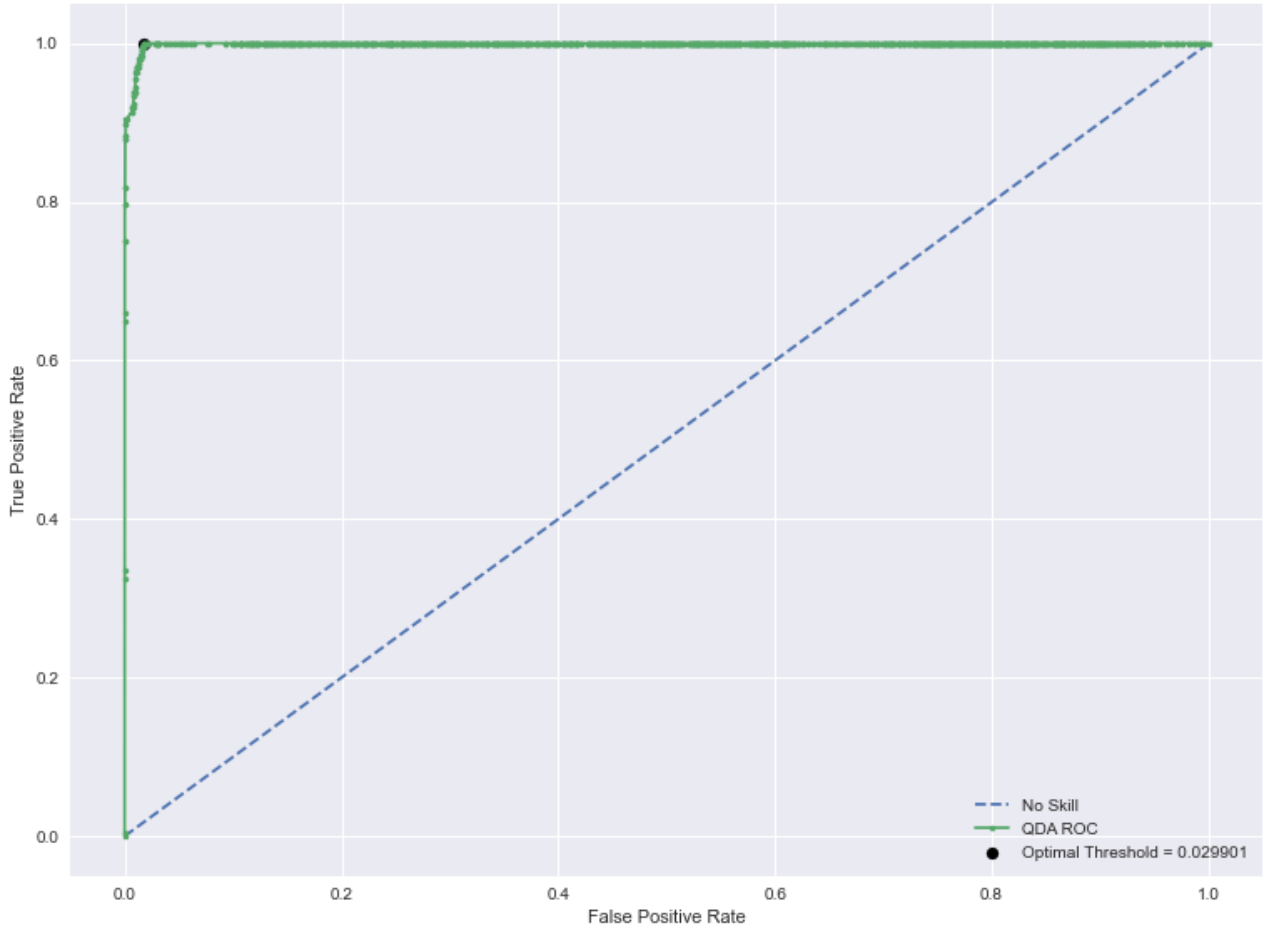
In [332…
```
roc_LDA = calculate_ROC(y_test, lda_probs, Type='LDA')
```

```
Best Threshold=0.003224
```



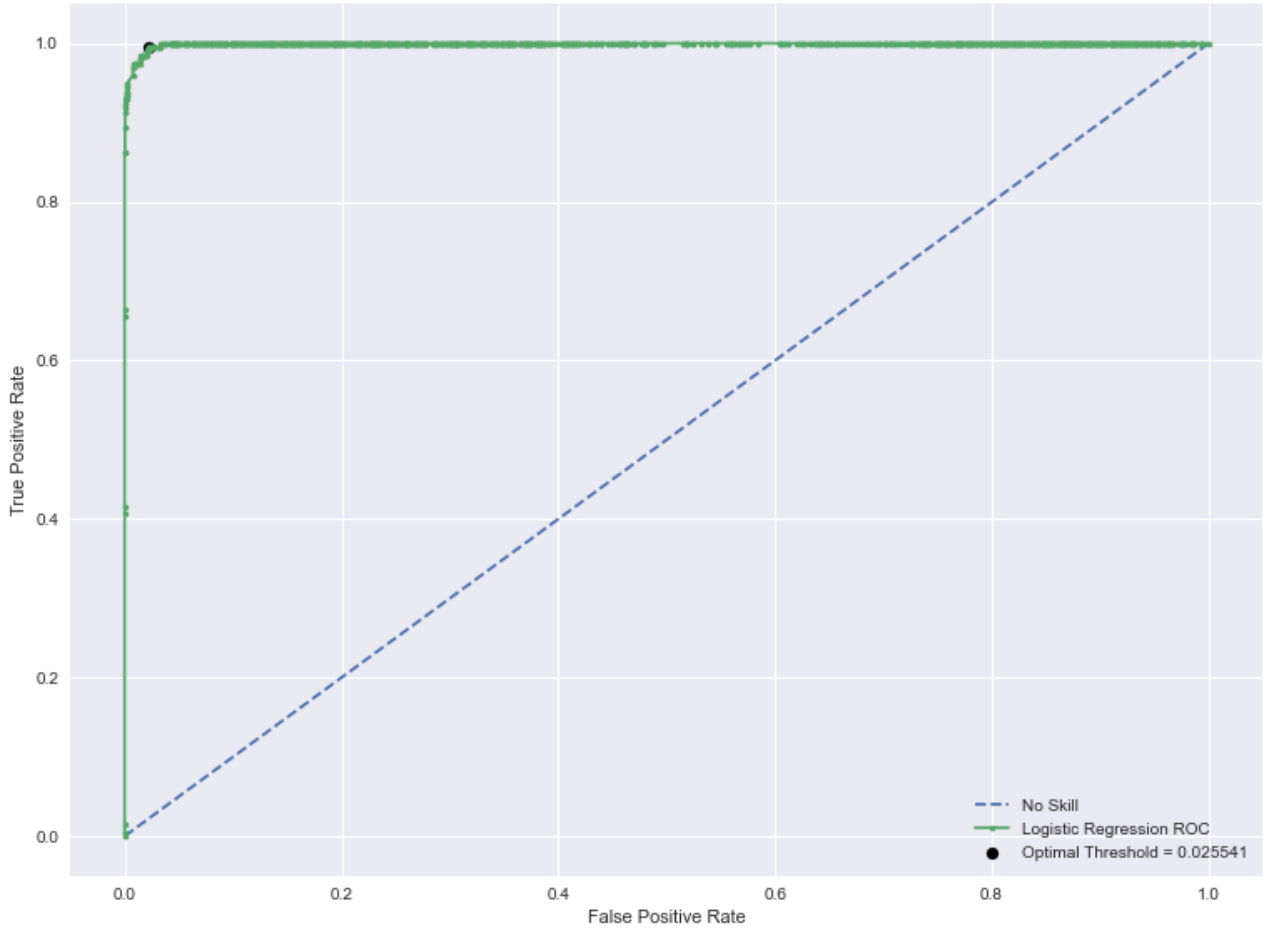
In [333…
```
roc_QDA = calculate_ROC(y_test, qda_probs, Type='QDA')
```

```
Best Threshold=0.029901
```



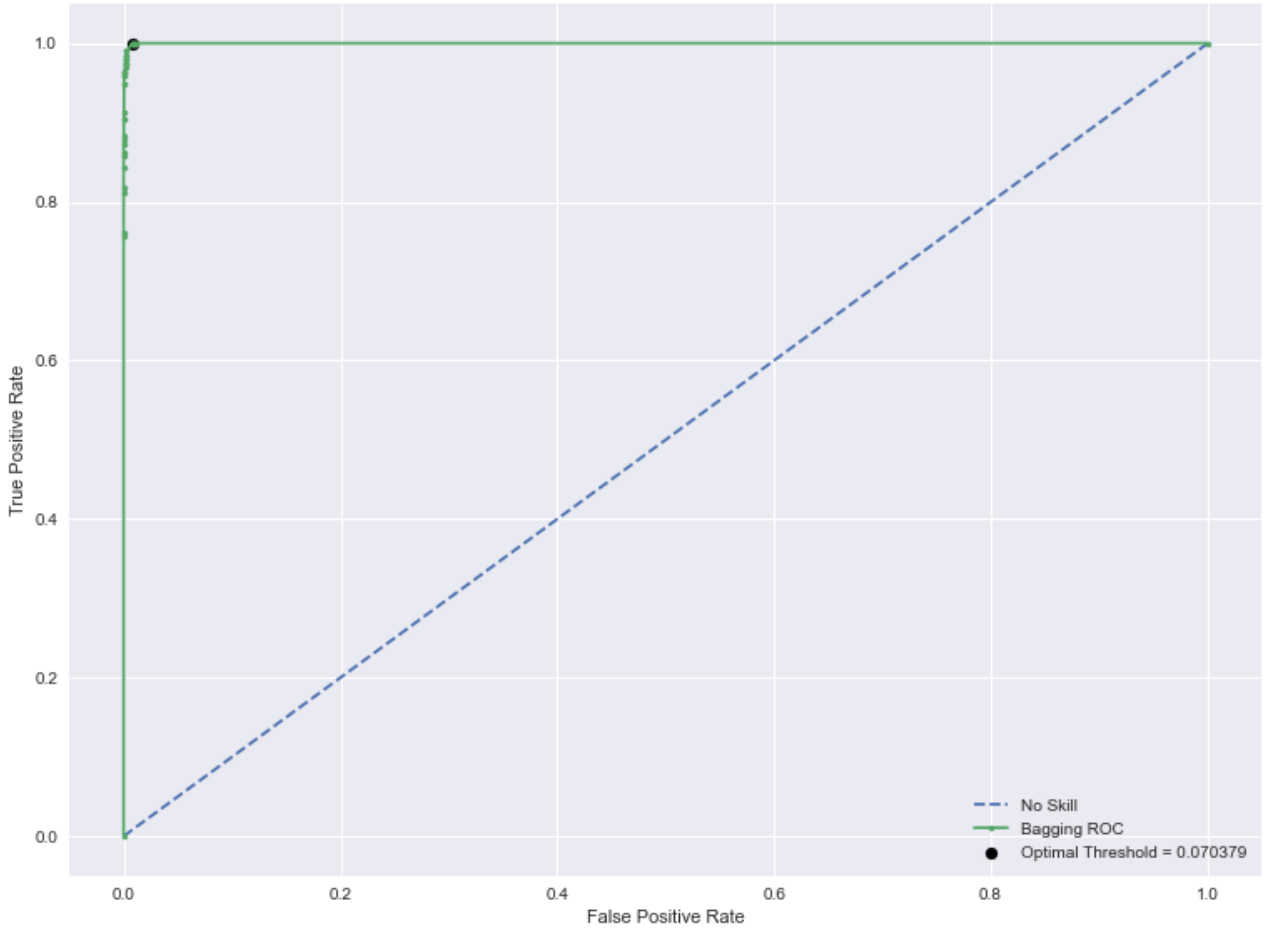
In [334…
```
roc_LogisticRegression = calculate_ROC(y_test, logreg_probs,Type='Logistic Regre
```

```
Best Threshold=0.025541
```



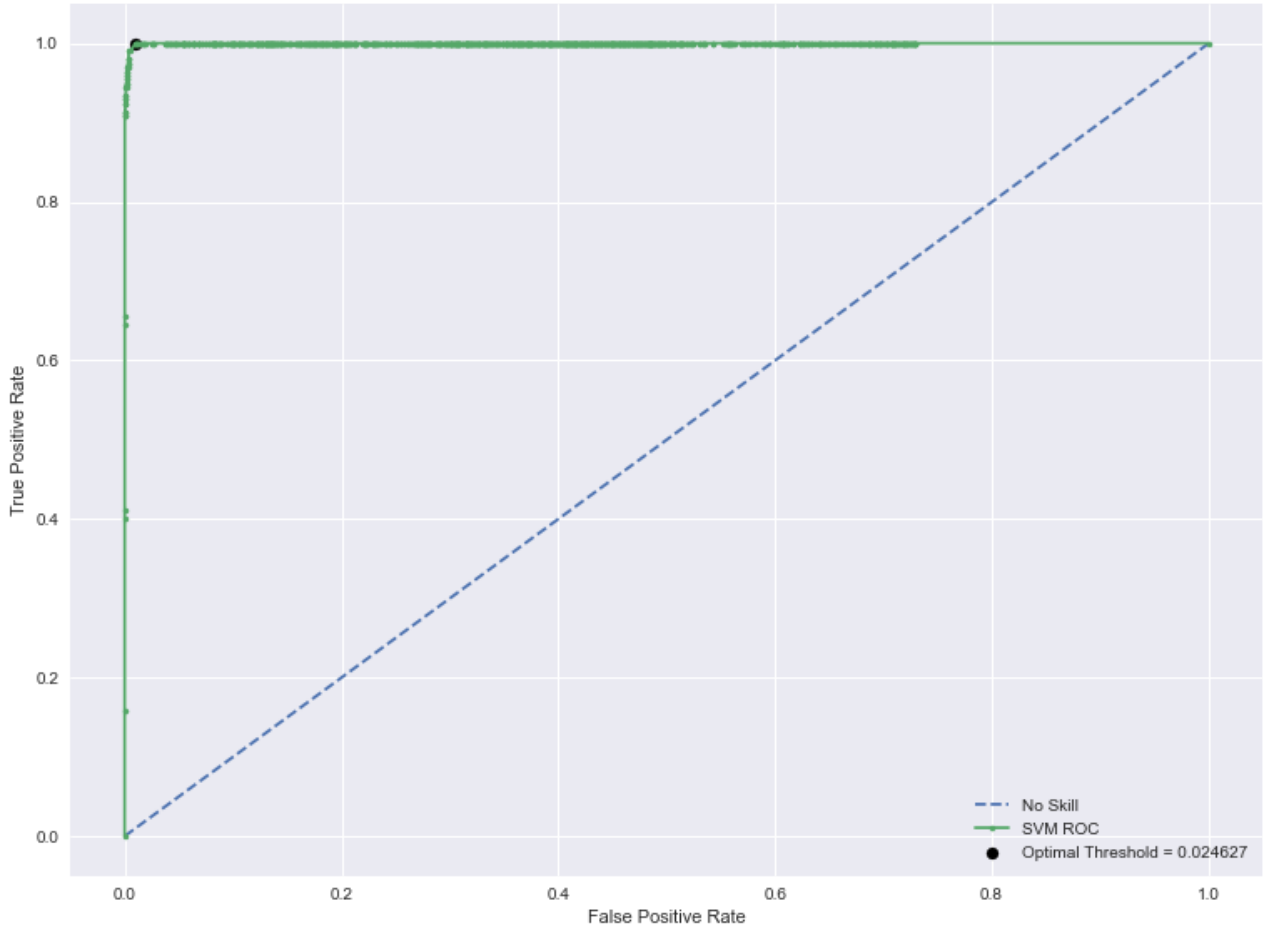
In [335…
```
roc_Bagging = calculate_ROC(y_test, bagging_probs,Type='Bagging')
```

```
Best Threshold=0.070379
```



In [336…
```
roc_SVM = calculate_ROC(y_test, clf_svm_probs,Type='SVM')
```

```
Best Threshold=0.024627
```



Confusion Matrix

In [337…

```
def conf_m(y_test,pred):
    """The confusion_matrix() function can be used to produce a confusion matrix
    in order to determine how many observations were correctly or incorrectly
    classified."""
    # print(confusion_matrix(y_test, pred).T)
    #ValueError: multilabel-indicator is not supported
    ##https://stackoverflow.com/questions/46953967/multilabel-indicator-is-not-s
    # print(confusion_matrix(y_test.argmax(), pred.argmax(axis)).T)

    #or, written this better way:
    conf_m = pd.DataFrame(confusion_matrix(y_test, pred))
    conf_m.columns.name = 'Predicted'
    conf_m.index.name = 'True'
    conf_m
    print(conf_m)

    # https://medium.com/ai-in-plain-english/understanding-confusion-matrix-and-

    """The classification_report() function gives us some summary
    statistics on the classifier's performance:"""

    # precision, recall, fbeta_score, support = precision_recall_fscore_support(
    # print(classification_report(y_test, logreg_pred, digits=3))

    # print("Sensitivity=Recall=Power: {}\nSpecificity=1-FPR: {}\nFPR: {}\nPreci
    # print(Specificity)

    """
```

```
the rows represent Actual Classes and the columns represent Class Predicted
by the model. So the top left cell (483) represents the KNN model was succes
at identifying 483 observations correctly labeled Blue Tarp, while 15 were
labeled Rooftop, 5 were labeled Soil, 1 were labeled Various Non-Tarp, and 0
were labeled Vegetation
while those mis-classification numbers are low, I need to ensure that they
stay even lower because the consequences for mis-classification are severe
(people in haiti earthquake hiding under blue tarps will not be found)
"""
```

In [338…
```
#KNN
knn_confusion_matrix = conf_m(y_test,knn_pred)
knn_sensitivity = 0.9746192893 # 192/(192+5) TPR = Sensitivity = TP/(TP+FN) ...
knn_specificity = 0.999510364 #1-0.0016323969425752 92 Specificity = 1 - FPR = T
knn_fpr = 0.000489636 # 3/(6124+3) = FPR = 1 - Specificity = FP/(TN+FP) ... Fals
knn_precision = 0.9846153846 # 192/(192+3) Precision = TruePositives / (TruePosi
```

```
Predicted     0    1
True
0          6126    1
1             7  190
```

In [339…
```
#LDA
lda_confusion_matrix = conf_m(y_test,lda_pred_scaled)
lda_sensitivity = 0.7969543147 # 157/(157+40) TPR = Sensitivity = TP/(TP+FN) ...
lda_specificity = 0.9890647952 #1-0.010844711569869995 = Specificity = 1 - FPR =
lda_fpr = 0.01093520483 # 67/(6060+67) = FPR = 1 - Specificity = FP/(TN+FP) ...
lda_precision = 0.7008928571 # 157/(157+67) Precision = TruePositives / (TruePos
```

```
Predicted     0    1
True
0          6060   67
1            40  157
```

In [340…
```
#QDA
qda_confusion_matrix = conf_m(y_test,qda_pred)
qda_sensitivity = 0.8730964467 # 172/(172+25) TPR = Sensitivity = TP/(TP+FN) ...
qda_specificity = 0.999510364 #1-0.00039197752662180704 = Specificity = 1 - FPR
qda_fpr = 0.0004896360372 # 6/(15301+6) = FPR = 1 - Specificity = FP/(TN+FP) ...
qda_precision = 0.9828571429 # 172/(172+3) Precision = TruePositives / (TruePosi
```

```
Predicted     0    1
True
0          6124    3
1            25  172
```

In [341…
```
#Logistic Regression
logistic_regression_confusion_matrix = conf_m(y_test,logreg_pred)
logreg_sensitivity = 0.923857868 # 182/(182+15) TPR = Sensitivity = TP/(TP+FN) .
logreg_specificity = 0.9990207279 #1-0.0009146142287842164 Specificity = 1 - FPR
logreg_fpr = 0.0009792720744 # 14/(15293+14) = FPR = 1 - Specificity = FP/(TN+FP
logreg_precision = 0.9680851064 # 182/(182+6) Precision = TruePositives / (TrueP
```

```
Predicted     0    1
True
0          6121    6
1            15  182
```

In [342…
```
#Bagging
#TN FP
#FN T
bagging_confusion_matrix = conf_m(y_test,bagging_pred)
bagging_sensitivity = 0.9471428571428572 # 663/(663+37) TPR = Sensitivity = TP/(
```

```
bagging_specificity = 0.9985126425384234 #1-0.0009146142287842164 Specificity =
bagging_fpr = 0.0014873574615765599 # 30/(20140+30) = FPR = 1 - Specificity = FP/
bagging_precision = 0.9567099567099567 # 663/(663+30) Precision = TruePositives
```

```
Predicted     0    1
True
0          6123    4
1             7  190
```

In [343…

```
#SVM
svm_confusion_matrix = conf_m(y_test, clf_svm_pred)
svm_sensitivity = 0.9471428571428572 # 663/(663+37) TPR = Sensitivity = TP/(TP+F
svm_specificity = 0.9985126425384234 #1-0.0009146142287842164 Specificity = 1 -
svm_fpr = 0.0014873574615765599 # 30/(20140+30) = FPR = 1 - Specificity = FP/(TN+
svm_precision = 0.9567099567099567 # 663/(663+30) Precision = TruePositives / (T
```

```
Predicted     0    1
True
0          6117   10
1            10  187
```

Table 2

Method	KNN (k=8)	LDA	QDA	Logistic Regression	Random Forest (n_estimators=500, max_features=auto, max_depth=100)	SVM (cost=3.801, degree=1, gamma=1, kernel='rbf')
Accuracy	0.999	0.983	0.996	0.997	0.998	0.988
AUC	0.987	0.893	0.936	0.961	0.982	0.987
ROC	see section above	see section above	see section above	see section above	see section above	see section above
Threshold	0.25	0.0032	0.0299	0.0255	0.0704	0.0246
Sensitivity=Recall=Power	0.965	0.797	0.873	0.924	0.964	0.949
Specificity=1-FPR	0.999	0.973	0.999	0.999	0.999	0.998
FPR	0.0002	0.027	0.001	0.001	0.001	0.002
Precision=PPV	0.995	0.485	0.983	0.968	0.979	0.949

Holdout Data

This next section uses the holdout data as the training set and the entire haiti pixels data as the test set.

In [344…

```
pixels.head(), pixels.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 63241 entries, 0 to 63240
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Class   63241 non-null  int64
 1   Red     63241 non-null  int64
 2   Green   63241 non-null  int64
```

```
 3   Blue    63241 non-null  int64
dtypes: int64(4)
memory usage: 1.9 MB
```

Out[344…
```
(   Class  Red  Green  Blue
 0      0   64     67    50
 1      0   64     67    50
 2      0   64     66    49
 3      0   75     82    53
 4      0   74     82    54,
 None)
```

In [345…
```
holdout.head(), holdout.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2008623 entries, 0 to 2008622
Data columns (total 4 columns):
 #   Column  Dtype
---  ------  -----
 0   Red     int64
 1   Green   int64
 2   Blue    int64
 3   Class   int64
dtypes: int64(4)
memory usage: 61.3 MB
```

Out[345…
```
(   Red  Green  Blue  Class
 0  104     89    63      0
 1  101     80    60      0
 2  103     87    69      0
 3  107     93    72      0
 4  109     99    68      0,
 None)
```

Here I am going to call the load_data() function.

In [346…
```
# the pixels dataset will be used for testing
X_pixels, y_pixels = load_data(pixels)

#the holdout dataset will be used for training
X_holdout, y_holdout = load_data(holdout)
```

There is no use calling the kfold_train_test_split() I wrote because the holdout data is just going to be assigned to the training set and the pixels data is going to be assigned to the test set.

In [348…
```
# the pixels dataset will be used for testing
X_test_pixels, y_test_pixels = X_pixels, y_pixels

#the holdout dataset will be used for training
X_train_holdout, y_train_holdout = X_holdout, y_holdout
```

We still need to scale our data to help prevent data leakage and to reduce the run time on the SVM model.

In [349…
```
X_train_holdout_scaled, X_test_pixels_scaled = scale_X_data(X_train_holdout, X_t
```

Now we can create the Random Forest model using the big training set. I am also going to calculate how long it takes to fit this model.

In [350…
```
import time # Just to compare fit times
start = time.time()

# using scaled X data for the bagging
```

```
bagging_pred_holdout, bagging_probs_holdout, rfc_holdout = bagging(X_train_holdo

end = time.time()
print("Tune Fit Time:", end - start)
```

```
Tune Fit Time: 474.6934726238251
```

In [390…
```
bagging_pred_holdout
```

Out[390…
```
63241
```

For this first time around we are going to call the eval_perform_rf() function to evaluate the performance of the Random Forest model that was just built, using the test data from haiti pixels set. We are going to feed it unscaled data from the pixels set.

In [351…
```
# to compare fit times
start = time.time()

eval_perform_rf(rfc_holdout, X_pixels, y_pixels, y_test_pixels, bagging_pred_hol

end = time.time()
print("Tune Fit Time:", end - start)
```

```
CV=1: 1.0
CV=2: 0.9999482470298647
CV=3: 0.9975247524752475
CV=4: 1.0
CV=5: 1.0
CV=6: 1.0
CV=7: 0.9868721313490382
CV=8: 0.9940132325875515
CV=9: 0.9081534378527693
CV=10: 0.9993541574954349


=== Confusion Matrix ===
[[61214     5]
 [  611  1411]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       0.99      1.00      0.99     61219
           1       1.00      0.70      0.82      2022

    accuracy                           0.99     63241
   macro avg       0.99      0.85      0.91     63241
weighted avg       0.99      0.99      0.99     63241



=== All AUC Scores ===
[1.         0.99994825 0.99752475 1.         1.         1.
 0.98687213 0.99401323 0.90815344 0.99935416]


=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9885865958789907
Tune Fit Time: 6.196226119995117
```

For 611 observations we predicted that it was not a blue tarp where it really was a blue tarp and only predicted 5 blue tarps that really were not blue tarps. While these are really good scores,

it's not ideal. I would actually like this to be the opposite: 5 False Negatives and 611 False Positives. The idea of people not being found outweighs thinking that something is a blue tarp when it really is not.

For our second time around, we will feed it scaled data from the holdout dataset.

In [352…
```
start = time.time()

eval_perform_rf(rfc_holdout, X_train_holdout_scaled, y_train_holdout, y_test_pix

end = time.time()
print("Tune Fit Time:", end - start)
```

```
CV=1: 1.0
CV=2: 1.0
CV=3: 1.0
CV=4: 1.0
CV=5: 0.9999999920308404
CV=6: 0.9976446365271635
CV=7: 0.9900376606542723
CV=8: 0.9741268696549095
CV=9: 0.9944207304130871
CV=10: 0.9866697081245687


=== Confusion Matrix ===
[[61214     5]
 [  611  1411]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       0.99      1.00      0.99     61219
           1       1.00      0.70      0.82      2022

    accuracy                           0.99     63241
   macro avg       0.99      0.85      0.91     63241
weighted avg       0.99      0.99      0.99     63241



=== All AUC Scores ===
[1.         1.         1.         1.         0.99999999 0.99764464
 0.99003766 0.97412687 0.99442073 0.98666971]


=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9942899597404841
Tune Fit Time: 522.2331917285919
```

Random Forest Hyperparameter Tuning

https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/

This guide was instrumental in developing my understanding of which hyperparameters really matter when tuning a random forest model. Initially, my approach was to just throw a bunch of parameters into the param_grid and let the model identify the optimal parameters through the use of GridSearchCV. But this quickly proved to be unscalable. I tried upgrading this approach

with RandomSearchCV but the big 2mil holdout dataset pummeled any hopes I had of using my original tuning function to fit the optimal random forest model. I needed to find a better solution to deal with large datasets.

In this guide from the analyticsvidhya.com article, I identified a few key parameters to focus on and learned how a much smaller range of options was necessary to pass to the gridsearch than I had initially thought. Here is a synopsis of my rationale for choosing the parameters that I did:

- **max_depth** determines the limit of the depth for each tree in the random forest. The performance of the model on training data increases continuously as max_depth increases because it gets closer and closer to a perfect fit of the data. Simultaneously, the fit on test data will decrease as max_depth increases since the model is overfit to the training data, and will thus perform poorly on the test data.
- **min_sample_split** determines the minimum number of observations for any node to split. By default this number is set to 2, which means that if any terminal node has more than two observations and is not a pure node, it can be split further into subnodes. As discussed above for max_depth, we want to avoid a random forest model comprised of trees with too many nodes, as that would indicate overfitting of the model. Leaving this min_sample_split number set to its default of 2 allows for this overfitting to occur, since 2 is so small and allows for trees to continue splitting until all the nodes are pure (1). "By increasing this number we can reduce the number of splits that happen in the decision tree and can thus prevent the model from overfitting" (or at least mitigate). However, it is important to not underfit this model (this is done by having the min_sample_split be too high, which would essentially lead to there being no significant splits observed, ultimately leading to a dip in both the training and test scores of model performance).
- **max_terminal_nodes / max_leaf_nodes** sets a condition on the splitting of the nodes in the tree, restricting the tree's growth. When the max_leaf_nodes is small, the random forest model will underfit.
- **min_samples_leaf** specifies the minimum number of samples that should be present in the leaf node after splitting a node. This is interesting to know but I'm not sure how helpful this will be in my model.
- **n_estimators** is super helpful for solving the exact problem I was having: as the number of trees used in the random forest model increases, so does the time complexity of the model. So, by limiting the n_estimators, we can control the time complexity from getting out of hand when working with large datasets.
- **max_samples** determines what fraction of the original dataset is given to any individual tree
- **max_features** determines the maximum number of features provided to each tree in the random forest model. The default value for this is set to the square root of the number of features present in the dataset. The ideal number of max_features generally tend to lie close to this value, so I will be leaving the max_features set to its default and will not be using it in the gridsearch.

In [353…

```
def rf_tune_hyperparameters(rfc_, X_train_rf_tune, y_train_rf_tune):
    #n_estimators determines the number of trees in the random forest
    n_estimators = [10, 50, 100]
```

```
# max_depth determines the maximum depth of the tree
max_depth = [4,8,12]

#min_samples_split determines the number of observations needed to split a n
min_samples_split = [5]

# create grid of these parameters that will be passed into the searchCV func
param_grid = {
 'n_estimators': n_estimators,
 'max_depth': max_depth,
 'min_samples_split': min_samples_split
 # 'max_features': max_features,
 }


# GridSearchCV of the parameters
rfc_gridsearch = GridSearchCV(
    rfc_,
    param_grid,
    # verbose=2,
    n_jobs=-1
    )

# # Random search of parameters
# rfc_random = RandomizedSearchCV(
#     estimator = model,
#     param_distributions = param_grid,
#     n_iter = 100,
#     cv = 3,
#     verbose=2,
#     random_state=313, n_jobs = -1)



# Fit the model
rfc_gridsearch.fit(X_train_rf_tune, y_train_rf_tune)


# rfc_random.fit(X_train.to_numpy(), y_train.to_numpy())
# this is a pretty data intensive model fit which is why i have set n_jobs t
#in the RandomizedSearchCV(), which means that all of my processors are bein
#used in parallel to run this fit

# print results
# print(rfc_random.best_params_)



#now store the optimal parameters and then return them for use in other func
optimal_n_estimators = rfc_gridsearch.best_params_['n_estimators']
optimal_max_depth = rfc_gridsearch.best_params_['max_depth']
optimal_min_samples_split = rfc_gridsearch.best_params_['min_samples_split']
# optimal_max_features = rfc_gridsearch.best_params_['max_features']



print("=== Optimal n_estimators ===")
print(optimal_n_estimators)
print('\n')
```

```
    print("=== Optimal optimal_max_depth ===")
    print(optimal_max_depth)
    print('\n')

    print("=== Optimal min_samples_split ===")
    print(optimal_min_samples_split)
    print('\n')

    # print("=== Optimal max_depth ===")
    # print(optimal_max_depth)
    # print('\n')

    # return rfc_random, optimal_n_estimators, optimal_max_features, optimal_max
    return rfc_gridsearch, optimal_n_estimators, optimal_max_depth, optimal_min_
```

In [355…

```
start = time.time()

rfc_gridsearch, optimal_n_estimators, optimal_max_depth, optimal_min_samples_spl

end = time.time()
print("Tune Fit Time:", end - start)
```

```
=== Optimal n_estimators ===
10


=== Optimal optimal_max_depth ===
12


=== Optimal min_samples_split ===
5


Tune Fit Time: 933.3132219314575
```

Now we can plug these optimal parameter values back into the model to see if they improve our model performance.

In [356…

```
def rf_optimal(X_train_rf_optimal, X_test_rf_optimal, y_train_rf_optimal, y_test
    rfc_optimal = RandomForestClassifier(
        n_estimators=optimal_n_estimators,
        max_depth=optimal_max_depth,
        min_samples_split=optimal_min_samples_split,
        random_state=313,
        n_jobs=-1)
    rfc_optimal.fit(X_train_rf_optimal, y_train_rf_optimal)
    rfc_optimal_predict = rfc_optimal.predict(X_test_rf_optimal)
    rfc_optimal_cv_score = cross_val_score(rfc_optimal, X_train_rf_optimal, y_tr
    for i in range(len(rfc_optimal_cv_score)):
        print('CV={}: {}'.format(i+1,rfc_optimal_cv_score[i]))
    print('\n')

    print("=== Confusion Matrix ===")
    print(confusion_matrix(y_test_rf_optimal, rfc_optimal_predict))
    print('\n')
    # === Confusion Matrix ===
    # [[15289    18]
    #  [   25   479]]

    print("=== Classification Report ===")
```

```
    print(classification_report(y_test_rf_optimal, rfc_optimal_predict))
    print('\n')

    print("=== All AUC Scores ===")
    print(rfc_optimal_cv_score)
    print('\n')

    print("=== Mean AUC Score ===")
    print("Mean AUC Score - Random Forest: ", rfc_optimal_cv_score.mean())

    return rfc_optimal_cv_score
```

In [358…

```
start = time.time()

rf_optimal_cv_score = rf_optimal(X_train_holdout_scaled, X_test_pixels_scaled, y

end = time.time()
print("Tune Fit Time:", end - start)
```

```
CV=1: 0.9999999256604689
CV=2: 1.0
CV=3: 0.9999999203505024
CV=4: 0.999999378405554
CV=5: 0.9999999973436134
CV=6: 0.9970048166131686
CV=7: 0.9913169949978118
CV=8: 0.9864831123087164
CV=9: 0.9951224951311811
CV=10: 0.9970546702323376


=== Confusion Matrix ===
[[61212     7]
 [  589  1433]]


=== Classification Report ===
              precision    recall  f1-score   support

           0       0.99      1.00      1.00     61219
           1       1.00      0.71      0.83      2022

    accuracy                           0.99     63241
   macro avg       0.99      0.85      0.91     63241
weighted avg       0.99      0.99      0.99     63241



=== All AUC Scores ===
[0.99999993 1.         0.99999992 0.99999938 1.         0.99700482
 0.99131699 0.98648311 0.9951225  0.99705467]


=== Mean AUC Score ===
Mean AUC Score - Random Forest:  0.9966981311043354
Tune Fit Time: 39.17140197753906
```

Support Vector Machines

Now we can create the SVM model for filling out Table 3, using the holdout dataset.

If I were to run the SVM on the +2mil holdout dataset, it would take forever to run. I let it run

overnight for 14 hours and when I checked in on it in the morning, it still had not completed. While it would be great to fit an SVM model to the entire dataset, that isn't feasible on my personal laptop. There are two options for me:

1. downsample the holdout dataset, fit an SVM model, report on the fit
2. use the model fit from the pixel dataset (above) and extrapolate those findings to this larger dataset

In [363…
```
def downsample_data_holdout(num_samples):
    #to ensure that i actually get 1000 samples for each category, i'll start by
    #splitting the data into two dataframes, one for Class=1 (blue tarp) and one
    #for Class=0 (not blue tarp)
    holdout_blue_tarp = holdout[y_holdout==1]
    holdout_not_blue_tarp = holdout[y_holdout==0]

    #let's see how this breaks down
    holdout_blue_tarp.shape # (18926, 4) ... 18926 observations
    holdout_not_blue_tarp.shape # (1989697, 4) ... 1989697 observations
    sum(y)/len(y) #0.03197292895431761
    #definitely an unbalanced dataset where there are way more examples of non-b
    #observations than there are blue tarp observations (this is expected)

    #now downsample the pixels_blue_tarp dataframe
    holdout_blue_tarp_downsampled = resample(holdout_blue_tarp,
                                             replace=False,
                                             n_samples=num_samples, #pass in num_
                                             random_state=313)

    #now downsample the pixels_not_blue_tarp dataframe
    holdout_not_blue_tarp_downsampled = resample(holdout_not_blue_tarp,
                                             replace=False,
                                             n_samples=num_samples, #pass in num_
                                             random_state=313)

    #now merge the two downsampled dataframes back into a single dataframe
    holdout_downsampled = pd.concat([holdout_blue_tarp_downsampled, holdout_not_
    #then print out the total number of samples (should be 2000 or 2*num_samples
    len(holdout_downsampled)

    return holdout_downsampled
```

In [368…
```
holdout_downsampled = downsample_data_holdout(num_samples=2000)
holdout_downsampled
```

Out[368…

	Red	Green	Blue	Class
2008514	99	135	160	1
2003629	72	73	88	1
2005410	111	117	153	1
2003223	124	143	195	1
2008058	78	84	90	1
...	...	...	...	...
1350109	112	119	83	0

	Red	Green	Blue	Class
1026534	255	255	230	0
76824	108	88	57	0
1660932	185	157	126	0
288801	48	43	32	0

4000 rows × 4 columns

In [370…
```
#split this data using the load_data() from above and pass it the downsampled da
X_downsampled_holdout, y_downsampled_holdout = load_data(holdout_downsampled)
X_downsampled_holdout, y_downsampled_holdout
```

Out[370…
```
(         Red  Green  Blue
 2008514   99    135   160
 2003629   72     73    88
 2005410  111    117   153
 2003223  124    143   195
 2008058   78     84    90
 ...      ...    ...   ...
 1350109  112    119    83
 1026534  255    255   230
 76824    108     88    57
 1660932  185    157   126
 288801    48     43    32

 [4000 rows x 3 columns],
 2008514    1
 2003629    1
 2005410    1
 2003223    1
 2008058    1
           ..
 1350109    0
 1026534    0
 76824      0
 1660932    0
 288801     0
 Name: Class, Length: 4000, dtype: int64)
```

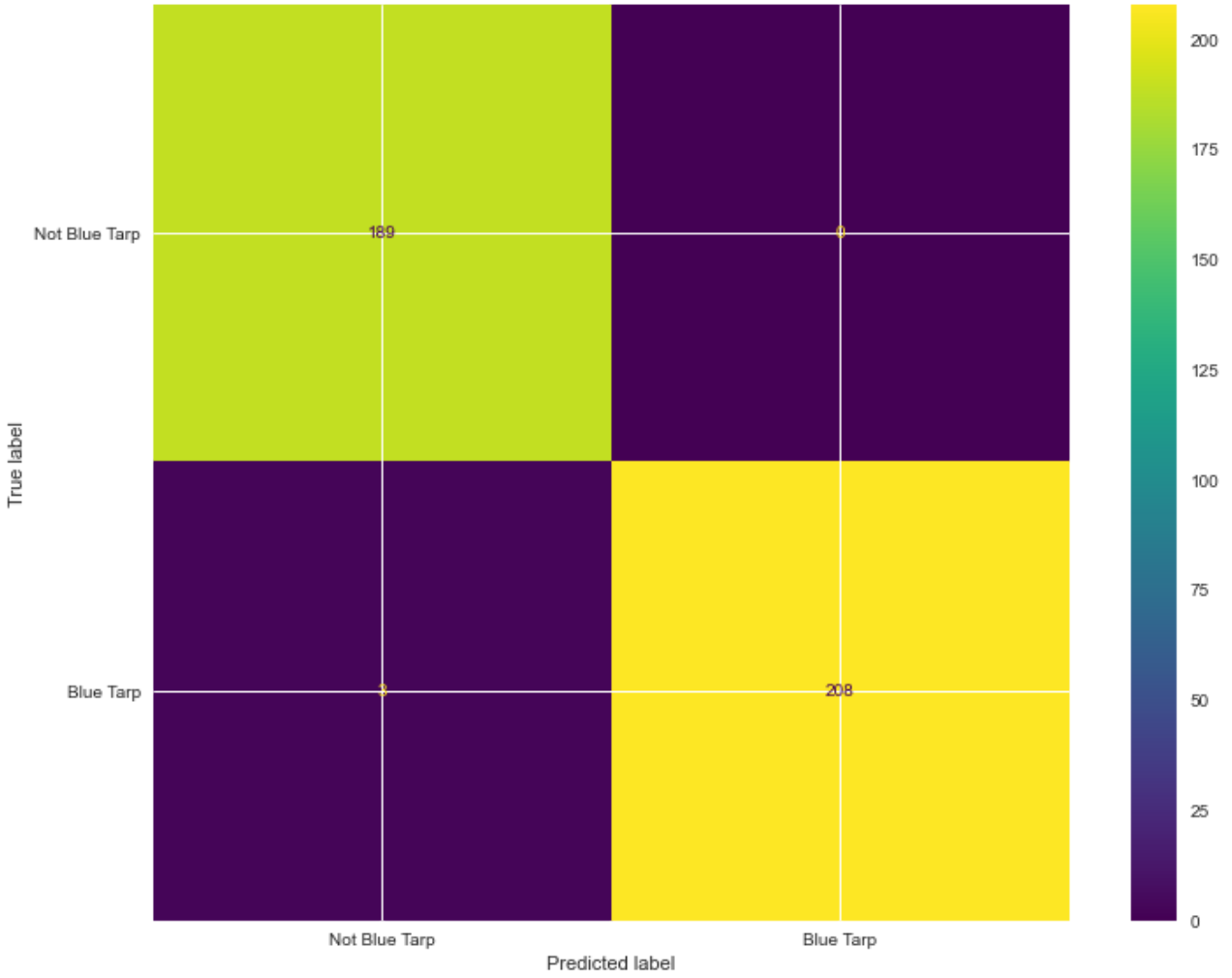
In [371…
```
X_train_downsampled_holdout, X_test_downsampled_holdout, y_train_downsampled_hol
```

In [372…
```
X_train_downsampled_holdout_scaled, X_test_downsampled_holdout_scaled = scale_X_
X_train_downsampled_holdout_scaled, X_test_downsampled_holdout_scaled
```

Out[372…
```
(array([[-0.34635117,  0.28982742,  0.54677391],
        [-0.91717256, -0.96246133, -0.57198924],
        [-0.09265277, -0.07374028,  0.43800527],
        ...,
        [-0.15607737, -0.65948825, -1.05367893],
        [ 1.47182068,  0.73418794,  0.01846909],
        [-1.42456936, -1.5684075 , -1.44213836]]),
 array([[ 1.40839609,  2.20865695,  2.02291974],
        [ 0.05533796, -0.11413669,  0.36031338],
        [-1.06516329, -0.82107389, -0.33891359],
        ...,
        [-0.09265277, -0.67968645, -0.99152542],
        [ 1.13355615,  0.71398974, -0.23014495],
        [-0.6211911 , -1.02305595, -1.10029406]]))
```

In [376…

```
# The build_svm() function builds the SVM model and can take in either downsampl
# but because of SVM's structure, it expects scaled data
def build_svm(X_train_foo_scaled, X_test_foo_scaled, y_train_foo, y_test_foo, cl
    # Now that we have our data split into test and train we can build our preli
    # support vector machine

    # clf_svm = SVC(random_state=313)
    clf_svm.fit(X_train_foo_scaled, y_train_foo)

    #that's it, that's our support vector machine for classification
    #now we can draw a confusion matrix and see how it performs on the test data

    plot_confusion_matrix(clf_svm,
                          X_test_foo_scaled,
                          y_test_foo,
                          values_format='d',
                          display_labels=["Not Blue Tarp", "Blue Tarp"])

#scaled data
build_svm(X_train_downsampled_holdout_scaled,X_test_downsampled_holdout_scaled,y
```



In [377…

```
#This performs incredibly well, which always makes me pause and think whether I
#The confusmion matrix shows us that of the 264 observations that were not blue
#I will use cross validation to optimize the parameters to see if I can improve

### SVM -- now we can actually build the SVM model
# use the svm() function I defined above
clf_svm_pred_downsampled, clf_svm_probs_downsampled, clf_svm_downsampled = svm(X
```

When we are optimizing a support vector machine we are attempting to find the best value for gamma and the regularization parameter C that will improve the prediction / classification accuracy of the model. Since we are looking to optimize two parameters (gamma and C) we will use GridSearchCV(), specifying several values for gamma and C and then letting the function determine the optimal combination (that's the great thing about using GridSerachCV() for problems like this: it tests all possible combinations of parameters for us and all we have to do is plug in some options).

The default values for the SVC parameters in sklearn.svm are:

- C=1.0
- kernel='rbf
- degree=3
- gamma='scale' (which is 1/(n_features*X.var()) while 'auto' uses 1/n_features

GridSearchSV in sklearn.model_selection has a parameter called param_grid where you can pass in a dictionary of parameters. I will pass in all of the default SVC parameters and then some extra parameters so that the GridSearchCV() can find the optimal combination from many options.

My assumption is that 'rbf' will work the best for the pixel data, so I might not need to have all these options for kernel, but I am including 'linear','poly', and 'sigmoid' anyway.

In [427…
```
def optimizing_svm():
    param_grid = [
      {'C': np.arange(0.001,5.001,.1),
       'degree':[1,2,3,4],
       'gamma': ['scale', 'auto', 1, 0.1, 0.01, 0.001, 0.0001],
       'kernel': ['rbf','linear','poly','sigmoid']},
    ]

    optimal_params = GridSearchCV(
        #SVC(),
        clf_svm_downsampled,
        param_grid,
        cv=5,
        scoring='accuracy',
        n_jobs=-1)

    return optimal_params

optimal_params_holdout_downsampled = optimizing_svm()
```

In [428…
```
optimal_params_holdout_downsampled
```

Out[428…
```
GridSearchCV(cv=5, estimator=SVC(probability=True, random_state=313), n_jobs=-1,
             param_grid=[{'C': array([1.000e-03, 1.010e-01, 2.010e-01, 3.010e-0
1, 4.010e-01, 5.010e-01,
       6.010e-01, 7.010e-01, 8.010e-01, 9.010e-01, 1.001e+00, 1.101e+00,
       1.201e+00, 1.301e+00, 1.401e+00, 1.501e+00, 1.601e+00, 1.701e+00,
       1.801e+00, 1.901e+00, 2.001e+00, 2.101e+00, 2.201e+00, 2.301e+00,
       2.401e+00...1e+00, 2.901e+00,
       3.001e+00, 3.101e+00, 3.201e+00, 3.301e+00, 3.401e+00, 3.501e+00,
       3.601e+00, 3.701e+00, 3.801e+00, 3.901e+00, 4.001e+00, 4.101e+00,
```

```
        4.201e+00, 4.301e+00, 4.401e+00, 4.501e+00, 4.601e+00, 4.701e+00,
        4.801e+00, 4.901e+00]),
                          'degree': [1, 2, 3, 4],
                          'gamma': ['scale', 'auto', 1, 0.1, 0.01, 0.001,
                                    0.0001],
                          'kernel': ['rbf', 'linear', 'poly', 'sigmoid']}],
                scoring='accuracy')
```

In [429…

```
import time # Just to compare fit times
start = time.time()

optimal_params_holdout_downsampled.fit(X_train_downsampled_scaled, y_train_downs

end = time.time()
print("Tune Fit Time:", end - start)
```

```
Tune Fit Time: 1532.1394097805023
```

In [431…

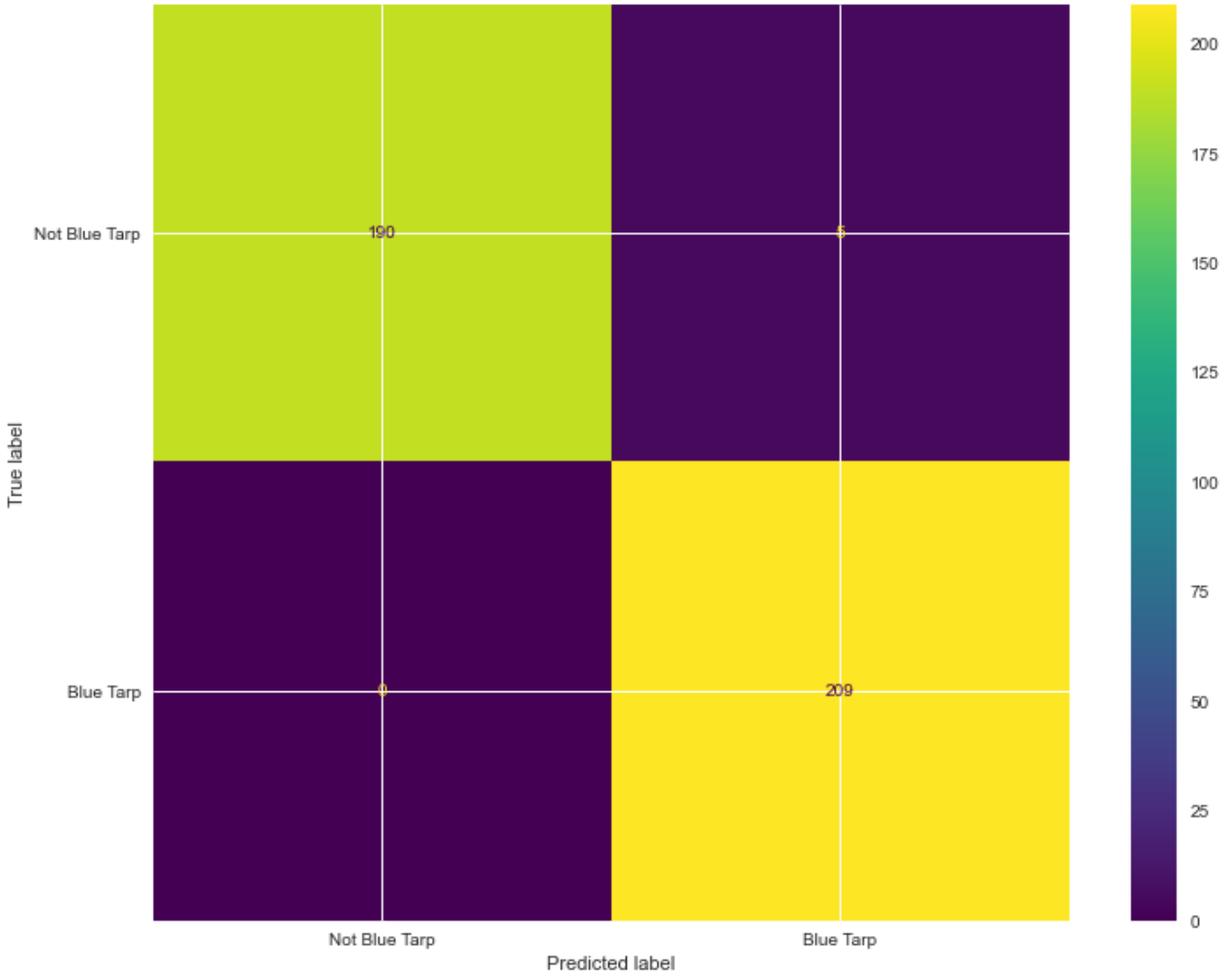
```
# Save the optimal parameters down in their own varaibles
best_C_holdout = optimal_params_holdout_downsampled.best_params_['C']
best_degree_holdout = optimal_params_holdout_downsampled.best_params_['degree']
best_gamma_holdout = optimal_params_holdout_downsampled.best_params_['gamma']
best_kernel_holdout = optimal_params_holdout_downsampled.best_params_['kernel']

# Then print out the optimal parameters for the model
print('''The best cost for clf_svm is: {}
the best degree for clf_svm is: {}
the best gamma for clf_svm is: {}
the best kernel for clf_svm is: '{}'
      '''.format(
      best_C_holdout,
      best_degree_holdout,
      best_gamma_holdout,
      best_kernel_holdout
))
```

```
The best cost for clf_svm is: 3.801
the best degree for clf_svm is: 1
the best gamma for clf_svm is: 1
the best kernel for clf_svm is: 'rbf'
```

This is the same as the pixels data...which makes me skeptical.

In [432…

```
# Now we can use these optimal parameters to fit a new model:
clf_svm_downsampled_optimal = SVC(random_state=313, C=best_C_holdout, degree=bes

build_svm(X_train_downsampled_scaled, X_test_downsampled_scaled, y_train_downsam
```



Completing Table 3

In [385…
```
#run all of the models with the holdout data
knn_pred_scaled, knn_probs_scaled = knn(X_train_holdout_scaled, X_test_pixels_sc
lda_pred_scaled, lda_probs_scaled = lda(X_train_holdout_scaled, X_test_pixels_sc
qda_pred_scaled, qda_probs_scaled = qda(X_train_holdout_scaled, X_test_pixels_sc
logreg_pred_scaled, logreg_probs_scaled = logistic_regression(X_train_holdout_sc
```

```
KNN performs best when the classifier k=25
```

Accuracy

In [393…
```
def accuracy(y_test, pred, method):
    accuracy = accuracy_score(y_test, pred)
    print('{} Accuracy: {}'.format(method,str(accuracy)))
    print('{} Test Error: {}'.format(method, str(1 - accuracy )))
    """that's actually not very helpful because this is scoring across classes,
    isn't giving us a good view of the accuracy of the model...use this instead:

    return accuracy
    # ^^ there is a severe imbalance
    #https://machinelearningmastery.com/threshold-moving-for-imbalanced-classifi
    #need to move threshold to accommodate this imbalance
    #"It has been stated that trying other methods, such as sampling, without tr
    #Pages 72, Imbalanced Learning: Foundations, Algorithms, and Applications, 2
```

```
    #rfc_holdout, X_train_holdout_scaled, y_train_holdout, y_test_pixels, baggin
    X_test_holdout_scaled
knn_accuracy = accuracy(y_test_pixels,knn_pred_scaled,'KNN')
lda_accuracy = accuracy(y_test_pixels,lda_pred_scaled,'LDA')
qda_accuracy = accuracy(y_test_pixels,qda_pred_scaled,'QDA')
logreg_accuracy = accuracy(y_test_pixels,logreg_pred_scaled,'Logistic Regression
bagging_accuracy = accuracy(y_test_pixels, bagging_pred_holdout,'Bagging')
svm_accuracy = accuracy(y_test_downsampled_holdout, clf_svm_pred_downsampled, 'S
```

```
KNN Accuracy: 0.9921411742382316
KNN Test Error: 0.00785882576176844
LDA Accuracy: 0.9851994750241141
LDA Test Error: 0.01480052497588591
QDA Accuracy: 0.9895162948087475
QDA Test Error: 0.010483705191252524
Logistic Regression Accuracy: 0.9895321073354312
Logistic Regression Test Error: 0.010467892664568823
Bagging Accuracy: 0.9902594835628785
Bagging Test Error: 0.009740516437121483
SVM Accuracy: 0.9925
SVM Test Error: 0.007499999999999951
```

AUC

We can then use the roc_auc_score() function to calculate the true-positive rate and false-positive rate for the predictions using a set of thresholds that can then be used to create a ROC Curve plot.

In [395…

```
def calculate_AUC(y_test, prob):
    # calculate scores
    auc = roc_auc_score(y_test, prob)
    return auc

auc_KNN = calculate_AUC(y_test_pixels, knn_pred_scaled)
print('auc_KNN:', auc_KNN)
auc_LDA = calculate_AUC(y_test_pixels, lda_pred_scaled)
print('auc_QDA:',auc_LDA)
auc_QDA = calculate_AUC(y_test_pixels, qda_pred_scaled)
print('auc_QDA:', auc_QDA)
auc_LogisticRegression = calculate_AUC(y_test_pixels, logreg_pred_scaled)
print('auc_LogisticRegression:',auc_LogisticRegression)
auc_Bagging = calculate_AUC(y_test_pixels, bagging_pred_holdout)
print('auc_Bagging:',auc_Bagging)
auc_SVM = calculate_AUC(y_test_downsampled_holdout, clf_svm_pred_downsampled)
print('auc_SVM:',auc_SVM)
```

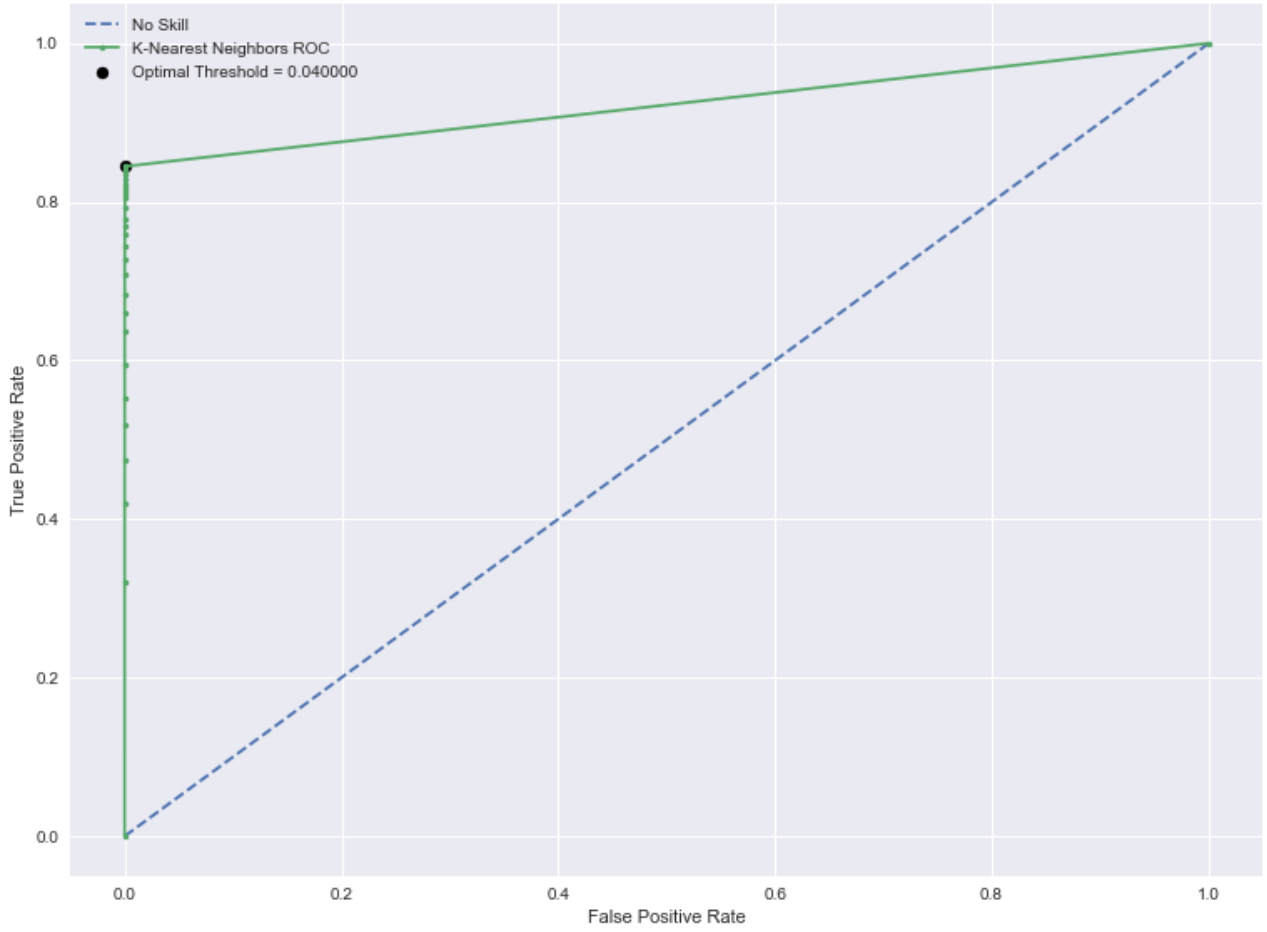
```
auc_KNN: 0.8792538920241414
auc_QDA: 0.8732772786400994
auc_QDA: 0.836053412462908
auc_LogisticRegression: 0.8363006923837784
auc_Bagging: 0.8488711313531196
auc_SVM: 0.9928909952606635
```

ROC

In [397…

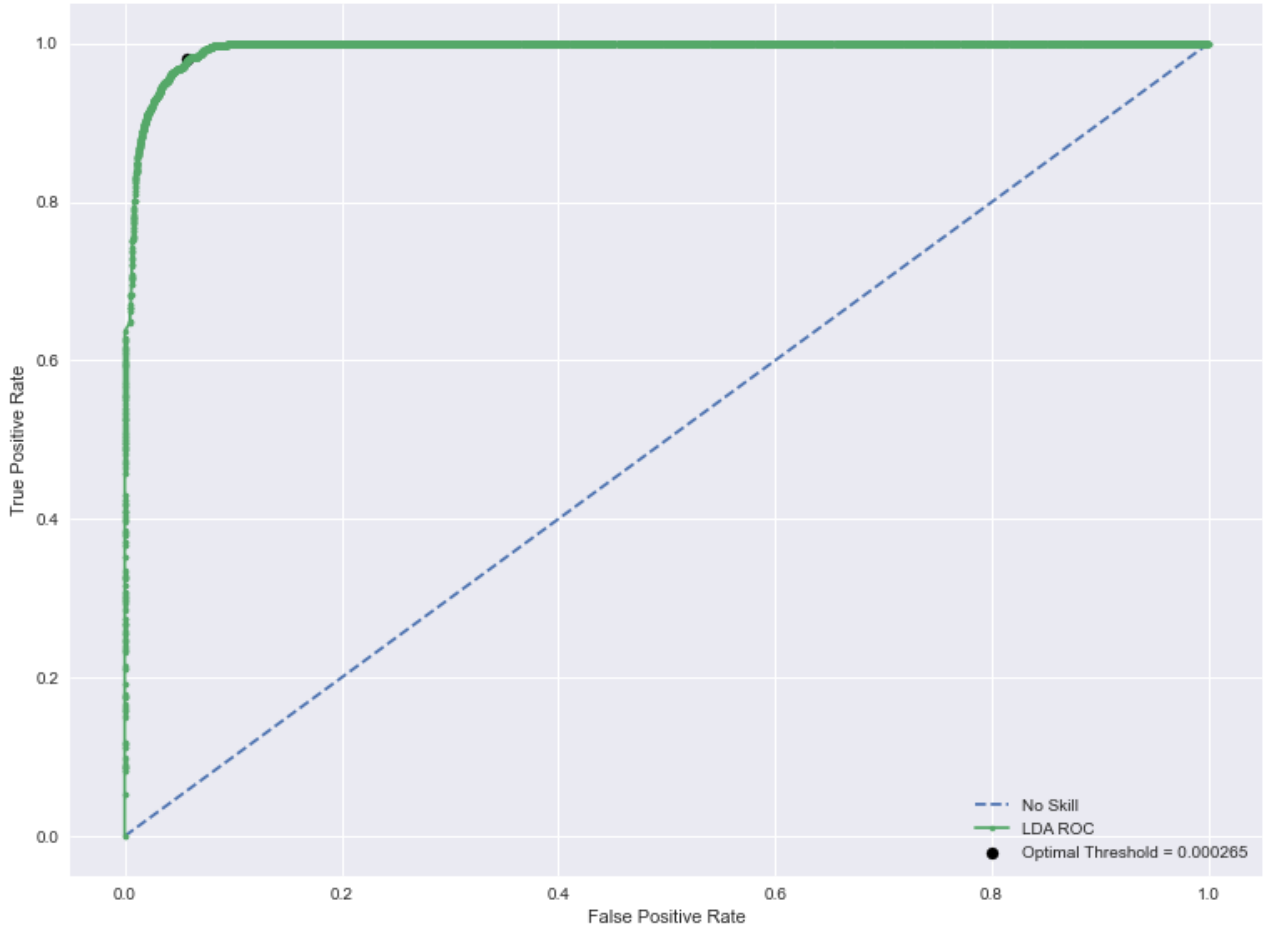
```
roc_KNN = calculate_ROC(y_test_pixels, knn_probs_scaled, Type='K-Nearest Neighbo
```

```
Best Threshold=0.040000
```



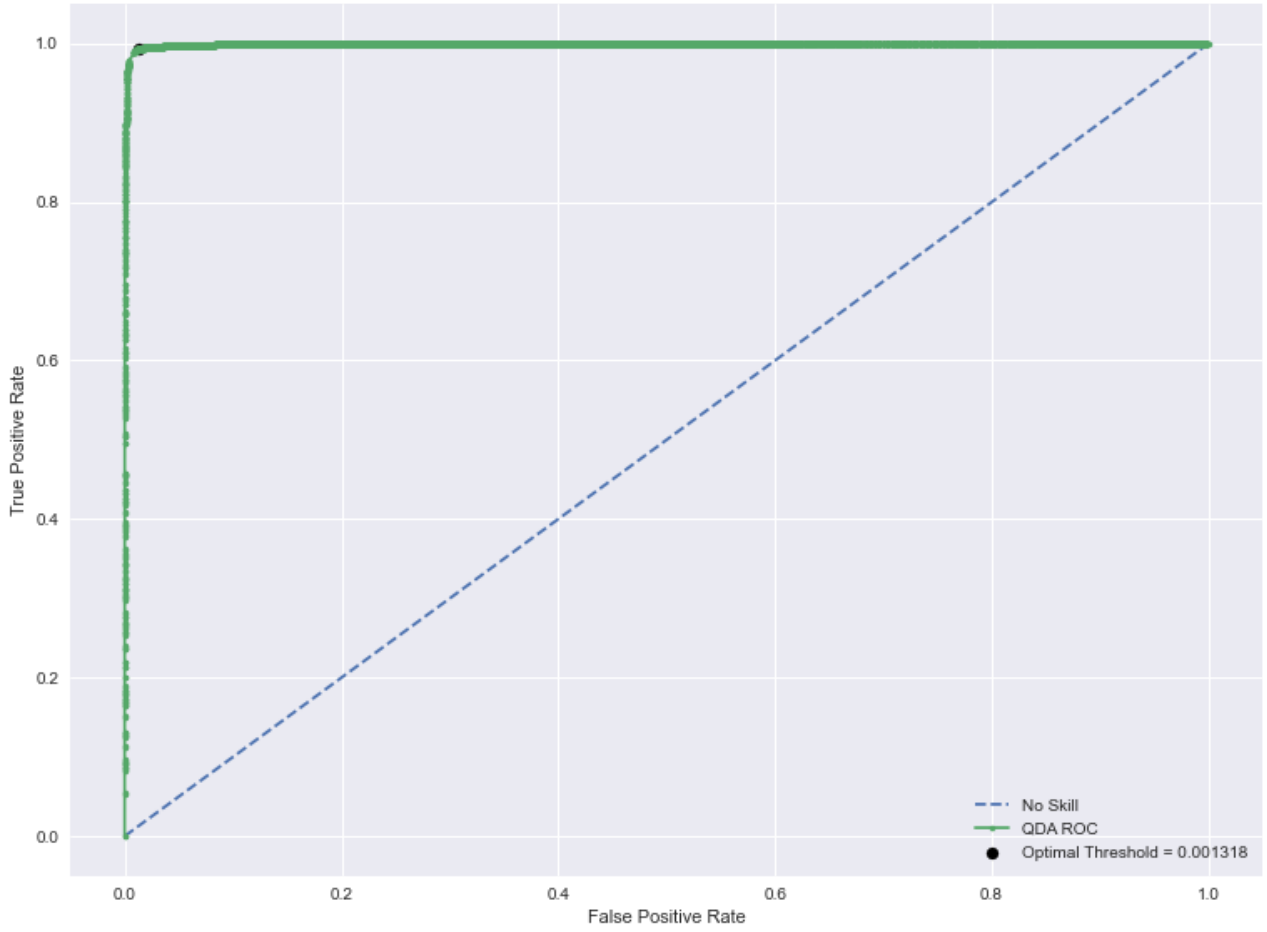
In [398…
```
roc_LDA = calculate_ROC(y_test_pixels, lda_probs_scaled, Type='LDA')
```

```
Best Threshold=0.000265
```



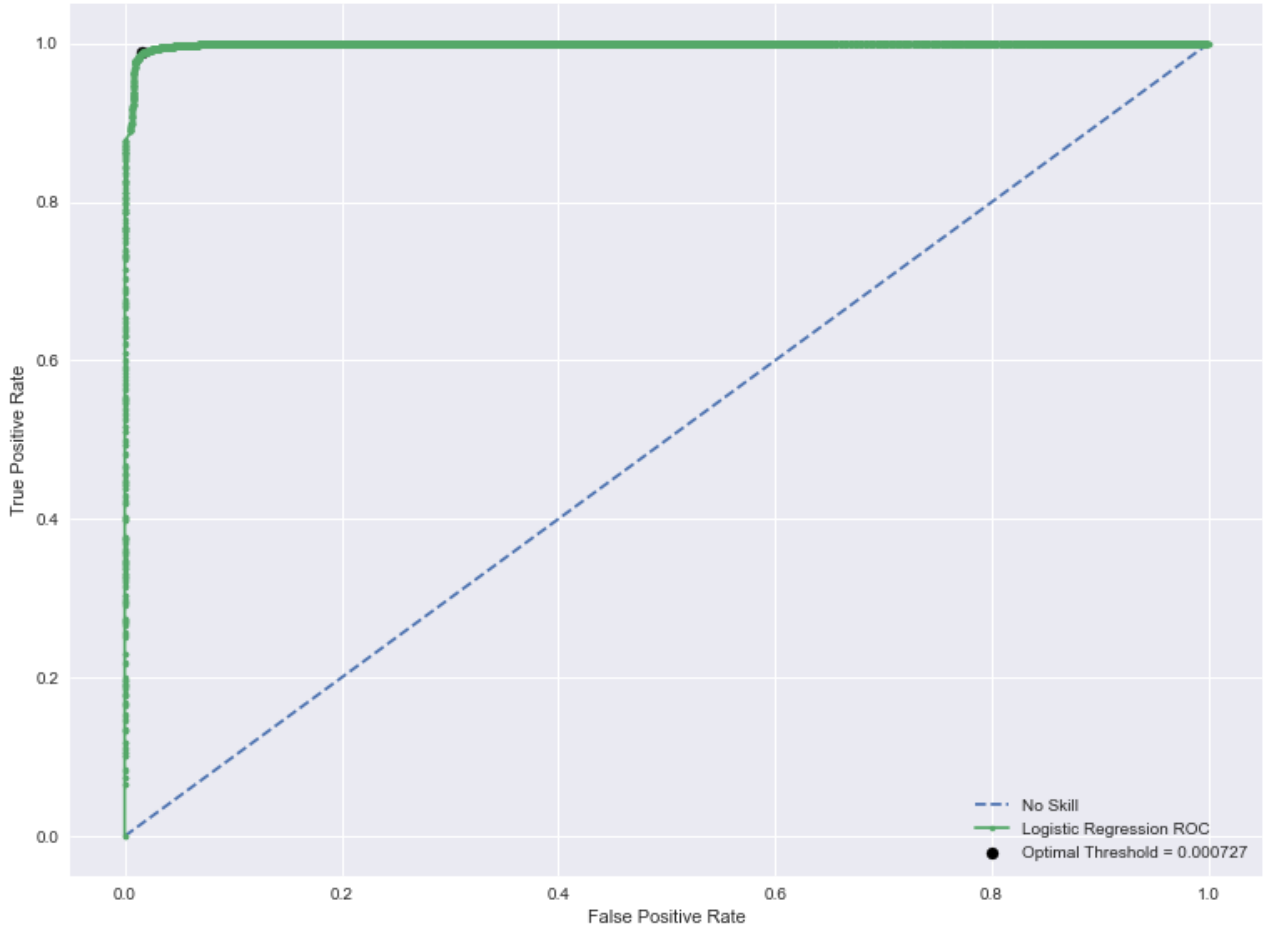
In [399…
```
roc_QDA = calculate_ROC(y_test_pixels, qda_probs_scaled, Type='QDA')
```

```
Best Threshold=0.001318
```



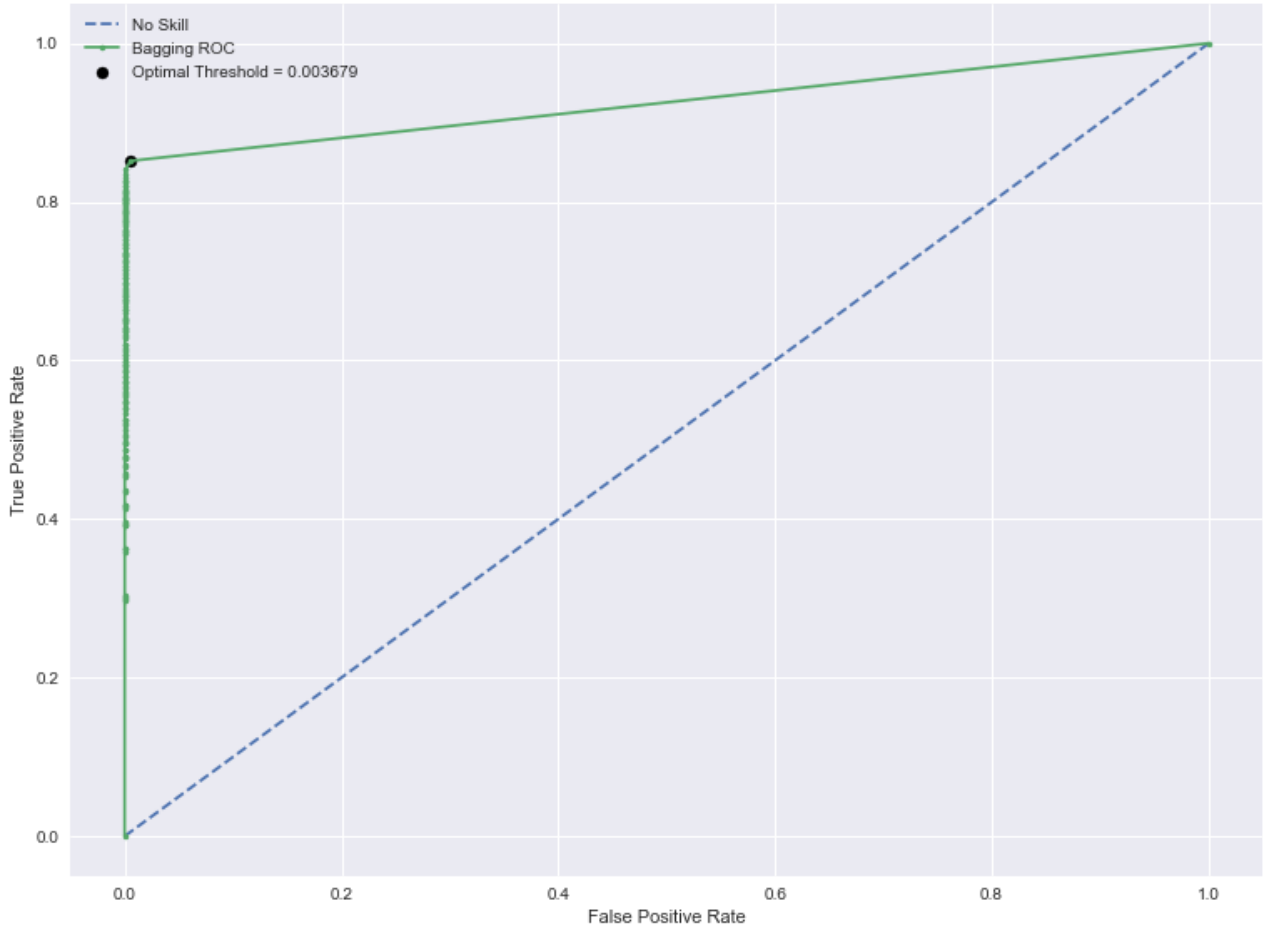
In [400…
```
roc_LogisticRegression = calculate_ROC(y_test_pixels, logreg_probs_scaled,Type='
```

```
Best Threshold=0.000727
```



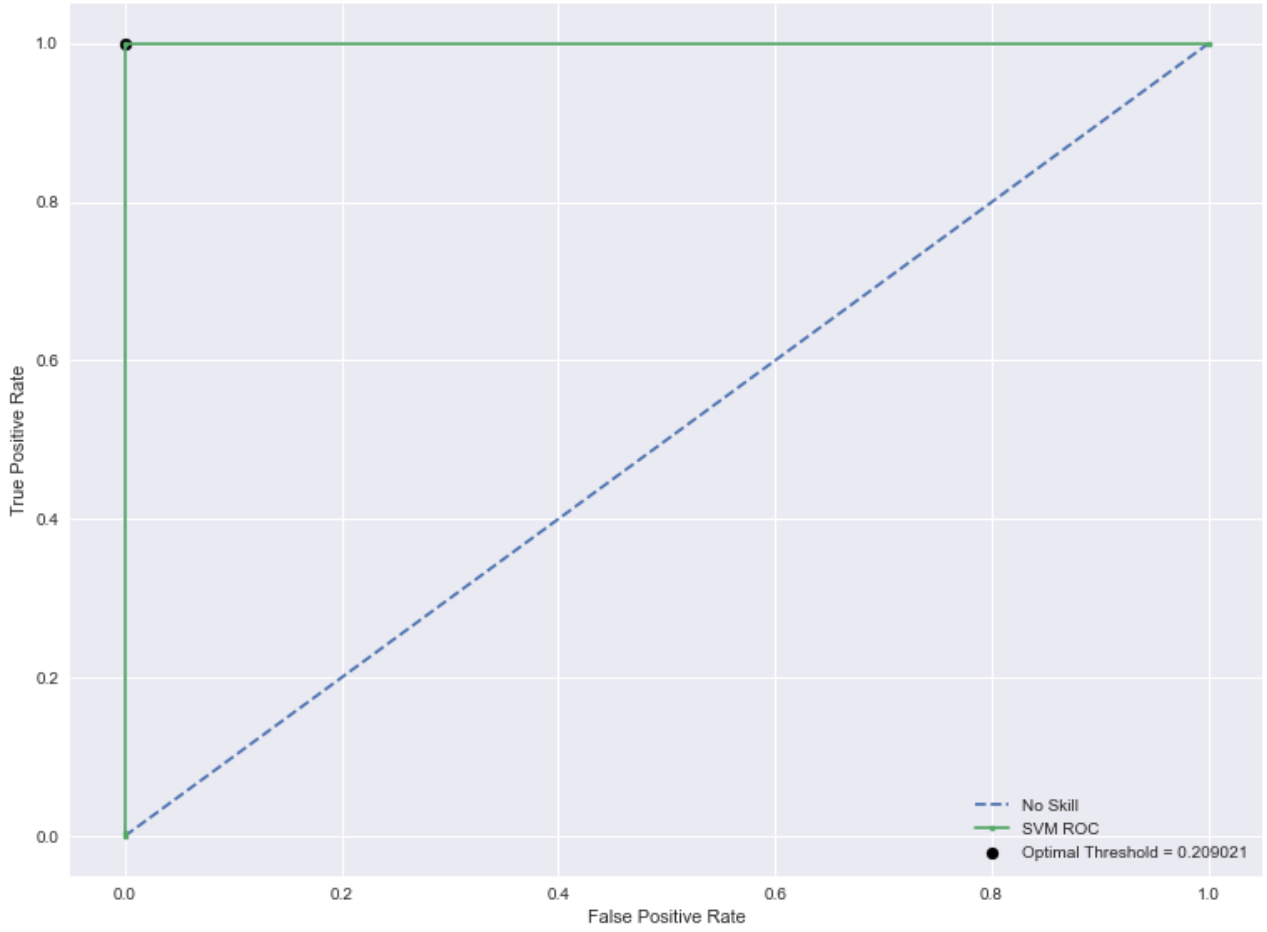
In [403…
```
roc_Bagging = calculate_ROC(y_test_pixels, bagging_probs_holdout,Type='Bagging')
```

```
Best Threshold=0.003679
```



In [404…
```
roc_SVM = calculate_ROC(y_test_downsampled_holdout, clf_svm_probs_downsampled,Ty
```

```
Best Threshold=0.209021
```



Confusion Matrix

In [408…
```
len(y_test_pixels), len(knn_pred),y_test_pixels,knn_pred
```

Out[408…
```
(63241,
 6324,
 0         0
 1         0
 2         0
 3         0
 4         0
         ..
 63236     1
 63237     1
 63238     1
 63239     1
 63240     1
 Name: Class, Length: 63241, dtype: int64,
 array([0, 0, 0, ..., 1, 1, 1]))
```

In [436…
```
#KNN
knn_confusion_matrix = conf_m(y_test_pixels,knn_pred_scaled)

print('\n')

tn = 61210
fp = 9
fn = 488
tp = 1534

sensitivity = tp / (tp + fn)
```

```
specificity = tn / (tn + fp)
fpr = 1-specificity
precision = tp / (tp + fp)

print('sensitivity:',sensitivity)
print('specificity:',specificity)
print('fpr:',fpr)
print('precision:',precision)
```

```
Predicted      0     1
True
0          61210     9
1            488  1534


sensitivity: 0.7586547972304649
specificity: 0.999852986817818
fpr: 0.00014701318218202086
precision: 0.9941672067401166
```

In [437…

```
#LDA
lda_confusion_matrix = conf_m(y_test_pixels,lda_pred_scaled)

print('\n')

tn = 60781
fp = 438
fn = 498
tp = 1524

sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
fpr = 1-specificity
precision = tp / (tp + fp)

print('sensitivity:',sensitivity)
print('specificity:',specificity)
print('fpr:',fpr)
print('precision:',precision)
```

```
Predicted      0     1
True
0          60781   438
1            498  1524


sensitivity: 0.7537091988130564
specificity: 0.9928453584671425
fpr: 0.007154641532857497 5
precision: 0.7767584097859327
```

In [438…

```
#QDA
qda_confusion_matrix = conf_m(y_test_pixels,qda_pred_scaled)

print('\n')

tn = 61219
fp = 0
fn = 663
tp = 1359

sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
```

```
fpr = 1-specificity
precision = tp / (tp + fp)

print('sensitivity:',sensitivity)
print('specificity:',specificity)
print('fpr:',fpr)
print('precision:',precision)
```

```
Predicted      0     1
True
0          61219     0
1            663  1359


sensitivity: 0.672106824925816
specificity: 1.0
fpr: 0.0
precision: 1.0
```

In [439…

```
#Logistic Regression
logistic_regression_confusion_matrix = conf_m(y_test_pixels,logreg_pred_scaled)

print('\n')

tn = 61219
fp = 0
fn = 662
tp = 1360

sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
fpr = 1-specificity
precision = tp / (tp + fp)

print('sensitivity:',sensitivity)
print('specificity:',specificity)
print('fpr:',fpr)
print('precision:',precision)
```

```
Predicted      0     1
True
0          61219     0
1            662  1360


sensitivity: 0.6726013847675568
specificity: 1.0
fpr: 0.0
precision: 1.0
```

In [440…

```
#Bagging
bagging_confusion_matrix = conf_m(y_test_pixels,bagging_pred_holdout)

print('\n')

tn = 61214
fp = 5
fn = 611
tp = 1411

sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
fpr = 1-specificity
```

```
precision = tp / (tp + fp)

print('sensitivity:',sensitivity)
print('specificity:',specificity)
print('fpr:',fpr)
print('precision:',precision)
```

```
Predicted      0     1
True
0          61214     5
1            611  1411


sensitivity: 0.6978239366963402
specificity: 0.9999183260098989
fpr: 8.167399010106102e-05
precision: 0.9964689265536724
```

In [435…

```
#SVM
svm_confusion_matrix = conf_m(y_test_downsampled_holdout, clf_svm_pred_downsampl

print('\n')
tn = 189
fp = 0
fn = 3
tp = 208

sensitivity = tp / (tp + fn)
specificity = tn / (tn + fp)
fpr = 1-specificity
precision = tp / (tp + fp)

print('sensitivity:',sensitivity)
print('specificity:',specificity)
print('fpr:',fpr)
print('precision:',precision)
```

```
Predicted    0    1
True
0          189    0
1            3  208


sensitivity: 0.985781990521327
specificity: 1.0
fpr: 0.0
precision: 1.0
```

Again, something strange is happening with my ROC curves. I wonder if it has to do with the downsampling of the data? I'm really not sure what explains the strange shapes and seeming loss of observations plotted in the ROC curves. There is no reason (at least that I can understand) for KNN, Random Forest, and SVM to perform worse than LDA, QDA, and Logistic Regression. It just doens't make sense. My assumption was that KNN, Random Forest, and SVM would blow the other models out of the water. Those three models make the most sense for solving a classification problem like this one, but the plots I've developed do not seem to support that. There is something missing here that I'm not quite able to put my finger on. If you could point me in the right direction, I would love to go back and correctly implement these models to accurately reflect what is happening with the data.

Table 3

Method	KNN (k=25)	LDA	QDA	Logistic Regression	Random Forest (n_estimators=10, max_depth=12, min_samples_split=5)	SVM (cost=3.801, degree=1, gamma=1, kernel='rbf')
Accuracy	0.999	0.983	0.996	0.997	0.990	0.993
AUC	0.987	0.893	0.936	0.961	0.849	0.993
ROC	see section above	see section above	see section above	see section above	see section above	see section above
Threshold	0.040	0.0003	0.001	0.001	0.004	0.209
Sensitivity=Recall=Power	0.759	0.754	0.672	0.673	0.69	0.986
Specificity=1-FPR	0.999	0.993	1.0	1.0	0.999	1.0
FPR	0.0001	0.027	0.0	0.001	0.000	0.0
Precision=PPV	0.994	0.777	1.0	1.0	0.996	1.0

Conclusions

I moved away from the multiclass approach in this submission despite continuing to have a strong feeling that this is, in some capacity, the correct approach. The way I see it, looking at this as a binary classification problem is too simplistic. If the goal is to provide humanitarian aid to displaced people who have found temporary shelter under blue tarps, the misclassification between a blue tarp and a roof seems less severe than a blue tarp and a tree or a body of water. There has been an earthquake and we need to get aid to people, so if the misclassification points to a rooftop as a tarp, and we go to the rooftop, I would imagine there is a greater liklihood of someone being at the rooftop location (could be the rooftop of a house, where people live) is higher than the liklihood of finding someone under a misclassified tree or, even more ridiculously, a body of water. Unfortunately, this probabilistic model is beyond my current understanding.

- This blogpost on the Tensorflow website seems like the direction I'd like to take this project, but I just did not feel comfortable implementing a solution when I don't fully understand the math behind it. https://blog.tensorflow.org/2019/03/regression-with-probabilistic-layers-in.html.
- I do, however, think this Haiti Pixels project would be an interesting thing to come back to when I'm farther along in my program to see if I can apply some of the new methodologies to this problem and achieve the level of success I think is possible from this type of problem.

"Random forests tend to shine in scenarios where a model has a large number of features that individually have weak predicative power but much stronger power collectively." https://medium.com/@hjhuney/implementing-a-random-forest-classification-model-in-python-583891c99652

Random Forests reduce this variance by creating decision trees for many subsets of the data and then taking the average of all those decision trees. In a typical Random Forest, a random subset of the features are chosen for each decision tree. In a bagging model, all of the features are chosen for each decision tree.

If I had decided to go with the multiclass approach, I think that the Random Forest model would have been the optimal classification method. But since I ended up treating this as a binary classification problem, SVM makes more sense and ultimately performed better. The only thing holding SVM back is the computational burden of running the model on such a large dataset. I would recommend using the SVM model for this problem with the important caveat that there be sufficient processing power to actually run the entire dataset or at least a substantially larger percentage for the downsampled data than I used.

Random Forest Hyperparameter Tuning

https://www.analyticsvidhya.com/blog/2020/03/beginners-guide-random-forest-hyperparameter-tuning/

This guide was instrumental in developing my understanding of which hyperparameters really matter when tuning a random forest model. Initially, my approach was to just throw a bunch of parameters into the param_grid and let the model identify the optimal parameters through the use of GridSearchCV. But this quickly proved to be unscalable. I tried upgrading this approach with RandomSearchCV but the big 2mil holdout dataset pummeled any hopes I had of using my original tuning function to fit the optimal random forest model. I needed to find a better solution to deal with large datasets.

In this guide from the analyticsvidhya.com article, I identified a few key parameters to focus on and learned how a much smaller range of options was necessary to pass to the gridsearch than I had initially thought. Here is a synopsis of my rationale for choosing the parameters that I did:

- **max_depth** determines the limit of the depth for each tree in the random forest. The performance of the model on training data increases continuously as max_depth increases because it gets closer and closer to a perfect fit of the data. Simultaneously, the fit on test data will decrease as max_depth increases since the model is overfit to the training data, and will thus perform poorly on the test data.
- **min_sample_split** determines the minimum number of observations for any node to split. By default this number is set to 2, which means that if any terminal node has more than two observations and is not a pure node, it can be split further into subnodes. As discussed above for max_depth, we want to avoid a random forest model comprised of trees with too many nodes, as that would indicate overfitting of the model. Leaving this min_sample_split number set to its default of 2 allows for this overfitting to occur, since 2 is so small and allows for trees to continue splitting until all the nodes are pure (1). "By increasing this number we can reduce the number of splits that happen in the decision tree and can thus prevent the model from overfitting" (or at least mitigate). However, it is important to not underfit this model (this is done by having the min_sample_split be too high, which would

essentially lead to there being no significant splits observed, ultimately leading to a dip in both the training and test scores of model performance).

- **max_terminal_nodes / max_leaf_nodes** sets a condition on the splitting of the nodes in the tree, restricting the tree's growth. When the max_leaf_nodes is small, the random forest model will underfit.
- **min_samples_leaf** specifies the minimum number of samples that should be present in the leaf node after splitting a node. This is interesting to know but I'm not sure how helpful this will be in my model.
- **n_estimators** is super helpful for solving the exact problem I was having: as the number of trees used in the random forest model increases, so does the time complexity of the model. So, by limiting the n_estimators, we can control the time complexity from getting out of hand when working with large datasets.
- **max_samples** determines what fraction of the original dataset is given to any individual tree
- **max_features** determines the maximum number of features provided to each tree in the random forest model. The default value for this is set to the square root of the number of features present in the dataset. The ideal number of max_features generally tend to lie close to this value, so I will be leaving the max_features set to its default and will not be using it in the gridsearch.

Concerns with using Bayesian modeling

Another idea I had to speed up the runtime was to use BayesSearchCV from scikit-optimize, which seemed promising at first but ended up not paning out the way I had hoped. First and foremost, I didn't feel comfortable implementing a solution when I did not fully understand the math behind it. Second, something about applying a Bayesian framework to a downsampled dataset seemed wrong to me. The idea of downsampling data is that you can apply the knowledge gained from a small subsample of a dataset and extrapolate the behavior seen in the subset to the larger dataset. And in this example, we are pushing that further by using the behavior seen in the holdout dataset to form conclusions about the pixels dataset. How could I, in good conscious, do this while using a theorem where the basic assumption is that your beliefs about data need to change when new data is available?

Said another way: model drift is the error that comes along with using a model fit on one dataset to then use that new data on the same model, because that new data might not fit the same distributional set. Bayesian ideology would partially solve for this but, as I just described, the problem is that using bayesian modeling on a downsampled dataset and then aplying it to a larger dataset seems to go directly against the idea of bayesian statistics: that you assumptions hould change when new data is in play.Said another way: model drift is the error that comes along with using a model fit on one dataset to then use that new data on the same model, because that new data might not fit the same distributional set. Bayesian ideology would partially solve for this but, as I just described, the problem is that using bayesian modeling on a downsampled dataset and then aplying it to a larger dataset seems to go directly against the idea of bayesian statistics: that you assumptions hould change when new data is in play.

Subsampling strategies in SVM ensembles

Patrick Koch and Wolfgang Konen

Cologne University of Applied Sciences
Institute of Computer Science
E-Mail: {patrick.koch, wolfgang.konen}@fh-koeln.de

Abstract

Support Vector Machines (SVMs) have shown to be strong methods for classification problems. Especially for difficult tasks the performance of SVMs is often superior to other learning algorithms. A main issue arising with this kernel-based learning is the high computation time and also the large memory demand required for training with large data. As a solution to this, ensemble-based SVM approaches have recently been proposed. Meyer *et al.* [1] investigated SVM ensembles based on bagging [2] and Cascade SVMs [3]. Stork *et al.* [4, 5] proposed ensembles based on boosting [6] and bagging with subsampling of the training data. In their experimental study they observed that subsampling is a necessary ingredient to impede overfitting. Unfortunately no rule-of-thumb could be given for the sample size parameter. The goal of this study is to get a deeper understanding which elements in a fruitful combination of individuals in SVM ensembles lead to considerable time savings while maintaining a good classification accuracy. First, we expect to obtain an asymptotic behaviour when we increase the ensemble size for a fixed training set size setting. Secondly, we want to measure the influence of the training set size on the classification accuracy. With these findings we try to give recommendations for sample size and ensemble size in order to balance computation time and accuracy. As a nice side effect, the observations made in this study can be used to create ensembles of other learning algorithms as well.

http://www.gm.fh-koeln.de/~konen/Publikationen/kochGMA2013.pdf

Concerns with downsampling

After reading this paper on Subsampling Strategies in SVM models, my main question here is whether or not there will be degredation in the predictive capacity of our model if we use a subsample of the original data that is a substantially smaller percentage to the original (consider something like 1000 observations from the 2mil observation dataset).

This paper (in section 3 page 9) tests different percentages (10, 25, 50) and compares them to the whole, to see how well they perform. The authors report only a slight degredation alongside

a substantial decrease in the computation time for the model, which is great news. The time complexity of running SVM on a large dataset is too taxing on my personal computer, so knowing that I can have confidence in the findings from my downsampled data (even if I'm using a very small percentage of the original data) is quite a relief. I am doing this project off of my laptop, which admittedly is pretty good, but I'm still only working with 6 cores and 32GB or RAM, so I can't afford to have a massively time and ddata intensive SVM run on my computer. I need speed because I need to be able to iterate quickly and determine the best model from the limited resources I have. So, downsampling my data must be the path forward since it offers me a "good enough" solution along with much improved computation times.

Something to consider here is the percentage that I'm using of the original holdout dataset. The paper I reference above mentions going as low as 10%, while I am going 0.1%. This is substantially less than discussed in the paper, so I may very well be taking such a small percentage of the dataset that I actually cannot trust the conclusions drawn from it because this is not a representative sample.