In [28]:
```
## Provide a wider display for easier viewing
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:100% !important; }</style>"))
```

In [2]:
```
## Remove future warnings for Pandas
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

## Import the necessary libraries
%matplotlib inline
import pandas as pd
import numpy as np
from sklearn import *
import seaborn as sns
import matplotlib.pyplot as plt
from scipy.stats import norm
import tensorflow as tf
from sklearn.multioutput import MultiOutputRegressor
from sklearn.model_selection import cross_val_score, train_test_split, GridSearchCV, StratifiedKFold, KFold
from sklearn.metrics import *
import category_encoders as ce
from sklearn.preprocessing import *
## from mlxtend.feature_selection import SequentialFeatureSelector as sfs
from sklearn.compose import TransformedTargetRegressor
```

Quation 1. (50 points) Use numeric prediction techniques to build a predictive model for the HW3.xlsx dataset. This dataset is provided on the course website and contains data about whether or not different consumers made a purchase in response to a test mailing of a certain catalog and, in case of a purchase, how much money each consumer spent. The data file has a brief description of all the attributes in a separate worksheet. Note that this dataset has two possible outcome variables: Purchase (0/1 value: whether or not the purchase was made) and Spending (numeric value: amount spent).

Dictionary

Codelist				
Var. #	**Variable Name**	**Description**	**Variable Type**	**Code Description**
1.	US	Is it a US address?	binary	1: yes 0: no
2 - 16	Source_*	Source catalog for the record	binary	1: yes 0: no
		(15 possible sources)		
17.	Freq.	Number of transactions in last year at source catalog	numeric	
18.	last_update_days_ago	How many days ago was last update to cust. record	numeric	
19.	1st_update_days_ago	How many days was 1st update to cust. record	numeric	
20.	Web_order	Customer placed at least 1 order via web	binary	1: yes 0: no
21.	Gender=mal	Customer is male	binary	1: yes 0: no
22.	Address_is_res	Address is a residence	binary	1: yes 0: no
23.	Purchase	Person made purchase in test mailing	binary	1: yes 0: no
24.	Spending	Amount spent by customer in test mailing ($)	numeric	

A. Data pre-processing and pre-analysis

1. Read in the data
2. Explore the features and target variables to assess what parameters will need to be changed
3. Prepare & transform data for data mining process

In [3]:

```
## Read in the data

purchases_df = pd.read_excel("HW3.xlsx")
```

In [4]:
```
## Observe the first five values to make sure data was read in correctly

purchases_df.head()
```

Out[4]:

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	...	source_x	source_w	Freq	last_
0	1	1	0	0	1	0	0	0	0	0	...	0	0	2	
1	2	1	0	0	0	0	1	0	0	0	...	0	0	0	
2	3	1	0	0	0	0	0	0	0	0	...	0	0	2	
3	4	1	0	1	0	0	0	0	0	0	...	0	0	1	
4	5	1	0	1	0	0	0	0	0	0	...	0	0	1	

5 rows × 25 columns

In [5]:
```
## Run summary statistics on every column except for the "Source", "Last Update", "First Update" columns
## Just to check for missing values

## Create a list of the columns we want to subset
summary_columns = ['US', 'Freq', 'Web order', 'Gender=male',
                   'Address_is_res', 'Purchase', 'Spending']

purchases_df[summary_columns].describe()
```

Out[5]:

	US	Freq	Web order	Gender=male	Address_is_res	Purchase	Spending
count	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000	2000.000000
mean	0.824500	1.417000	0.426000	0.524500	0.221000	0.500000	102.560745
std	0.380489	1.405738	0.494617	0.499524	0.415024	0.500125	186.749816
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	1.000000	1.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	1.000000	1.000000	0.000000	1.000000	0.000000	0.500000	1.855000
75%	1.000000	2.000000	1.000000	1.000000	0.000000	1.000000	152.532500
max	1.000000	15.000000	1.000000	1.000000	1.000000	1.000000	1500.060000

In [6]:
```
## We check to see what our "target" variable is - Purchase
## It looks like it is a binary classification problem.
## Since we're using numeric prediction, we can leave this the same

purchases_df.Purchase.unique()
```

Out[6]:
```
array([1, 0], dtype=int64)
```

In [7]:
```
## View splits for the targets

total_obvs = purchases_df.groupby("Purchase")["US"].count().sum()
total_purchases = purchases_df.groupby("Purchase")["US"].count()[0]
total_nopurchase = purchases_df.groupby("Purchase")["US"].count()[1]

## Evenly distributed right in the middle
print("{} total customers purchased, \
     {} % of overall observations".format(total_purchases, round(total_purchases/total_obvs*100, 2)))
print("{} total customers did not purchase, \
     {} % of overall observations".format(total_nopurchase, round(total_nopurchase/total_obvs*100, 2)))
```

```
1000 total customers purchased,      50.0 % of overall observations
1000 total customers did not purchase,      50.0 % of overall observations
```

We see an even split of the data. I don't see a need to stratify in this instance because of the split of the information. If they were skewed one way or the other then it might be needed.

Let's start building our models!

In [10]:
```
## Remove any columns that will not be used as features
## I am making a conscious decision to remove sequence_number - I believe it is extraneous and not needed
## It won't help the predictive power of the models

feature_cols = purchases_df.columns[~purchases_df.columns.isin(["sequence_number", "Purchase", "Spending"])]
feature_cols
```

Out[10]:
```
Index(['US', 'source_a', 'source_c', 'source_b', 'source_d', 'source_e',
       'source_m', 'source_o', 'source_h', 'source_r', 'source_s', 'source_t',
       'source_u', 'source_p', 'source_x', 'source_w', 'Freq',
       'last_update_days_ago', '1st_update_days_ago', 'Web order',
       'Gender=male', 'Address_is_res'],
      dtype='object')
```

In [11]:
```
## Create our "X" variable that contains all the features we are curious about plotting
X = np.array(purchases_df[feature_cols])

## Create our "y" variable which is our target variable
y = np.array(purchases_df["Spending"])

## Split data training 70 % and testing 30%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

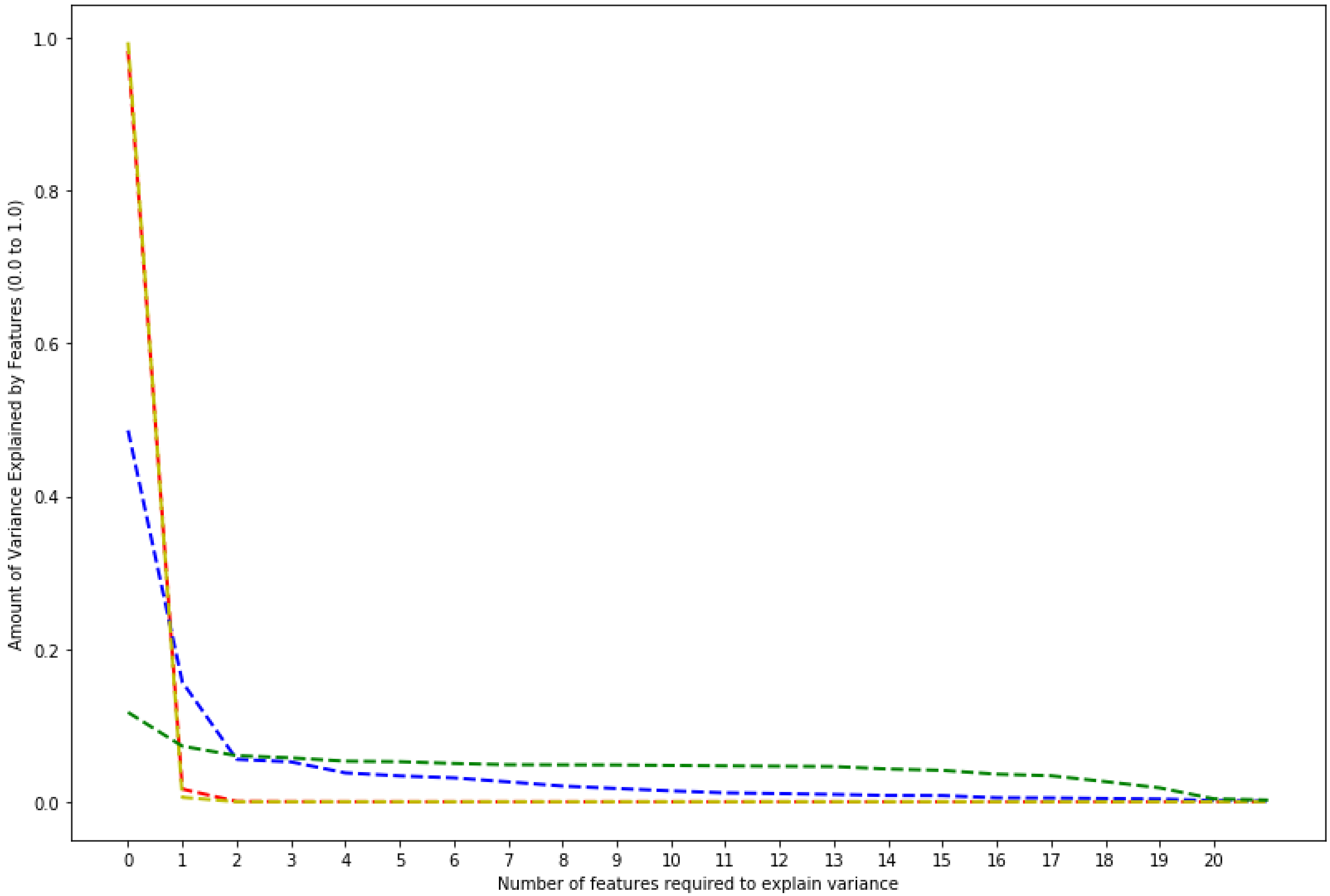
First thing I will do is assess which features are actually needed. To do this, I will use both a Principle Component Analysis, along with a Feature Importance Plot.

In [13]:

```
## We normalize our training and testing data PCA to work correctly
## We don't want to skew the results of the plot because some features are not on the same scale
## We perform this normalization AFTER splitting the data - again, so that we don't skew the training
## data with the testing data.

## Normalization is the process of scaling individual samples to have unit norm. This process can be useful
## if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pa
ir of samples.
X_train_norm = Normalizer(norm = "l1").fit_transform(X_train)
X_train_robust = RobustScaler().fit_transform(X_train)
X_train_power = PowerTransformer(method='yeo-johnson', standardize=False).fit_transform(X_train)
X_train_standard = StandardScaler().fit_transform(X_train)

## Import PCA from sklearn
from sklearn.decomposition import PCA

## Initialize two new PCA instances, we'll use this to plot the training data using two different transformations
## Normalizer will likely be skewed by the outliers in each of the features being used
## Robust is going to transform feature values to be larger than the previous scalers and more importantly are approxim
ately similar to original data
## PowerTransformer is a family of parametric, monotonic transformations that aim to map data from any distribution to
 as close to a Gaussian distribution
## as possible in order to stabilize variance and minimize skewness.

pca = PCA()
pca_r = PCA()
pca_p = PCA()
pca_s = PCA()

## The goal of this plot is to determine what features need to be included in our models
## In the first few homeworks, we threw the kitchen sink at the models.  Here, we are going to be
## more refined in our analysis.

## Train the PCA instance using the normalized training data
pca.fit(X_train_norm)
pca_r.fit(X_train_robust)
pca_p.fit(X_train_power)
pca_s.fit(X_train_standard)

plt.figure(1, figsize=(15, 10))
plt.clf()
plt.axes([.2, .2, .7, .7])

plt.plot(pca.explained_variance_ratio_, 'r--', linewidth = 2)
plt.plot(pca_r.explained_variance_ratio_, 'b--', linewidth = 2)
plt.plot(pca_p.explained_variance_ratio_, 'y--', linewidth = 2)
plt.plot(pca_s.explained_variance_ratio_, 'g--', linewidth = 2)
```

```python
## Set plot labels
plt.xlabel('Number of features required to explain variance')
plt.ylabel('Amount of Variance Explained by Features (0.0 to 1.0)')

## Explicitly set the x-axis data so we can see where the drop-off is
plt.xticks(np.arange(0, 21, step=1))

## Show the graph!
plt.show()
```

```
C:\Python\lib\site-packages\sklearn\utils\validation.py:595: DataConversionWarning: Data with input dtype int64 was co
nverted to float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
C:\Python\lib\site-packages\sklearn\utils\validation.py:595: DataConversionWarning: Data with input dtype int64 was co
nverted to float64 by StandardScaler.
  warnings.warn(msg, DataConversionWarning)
```



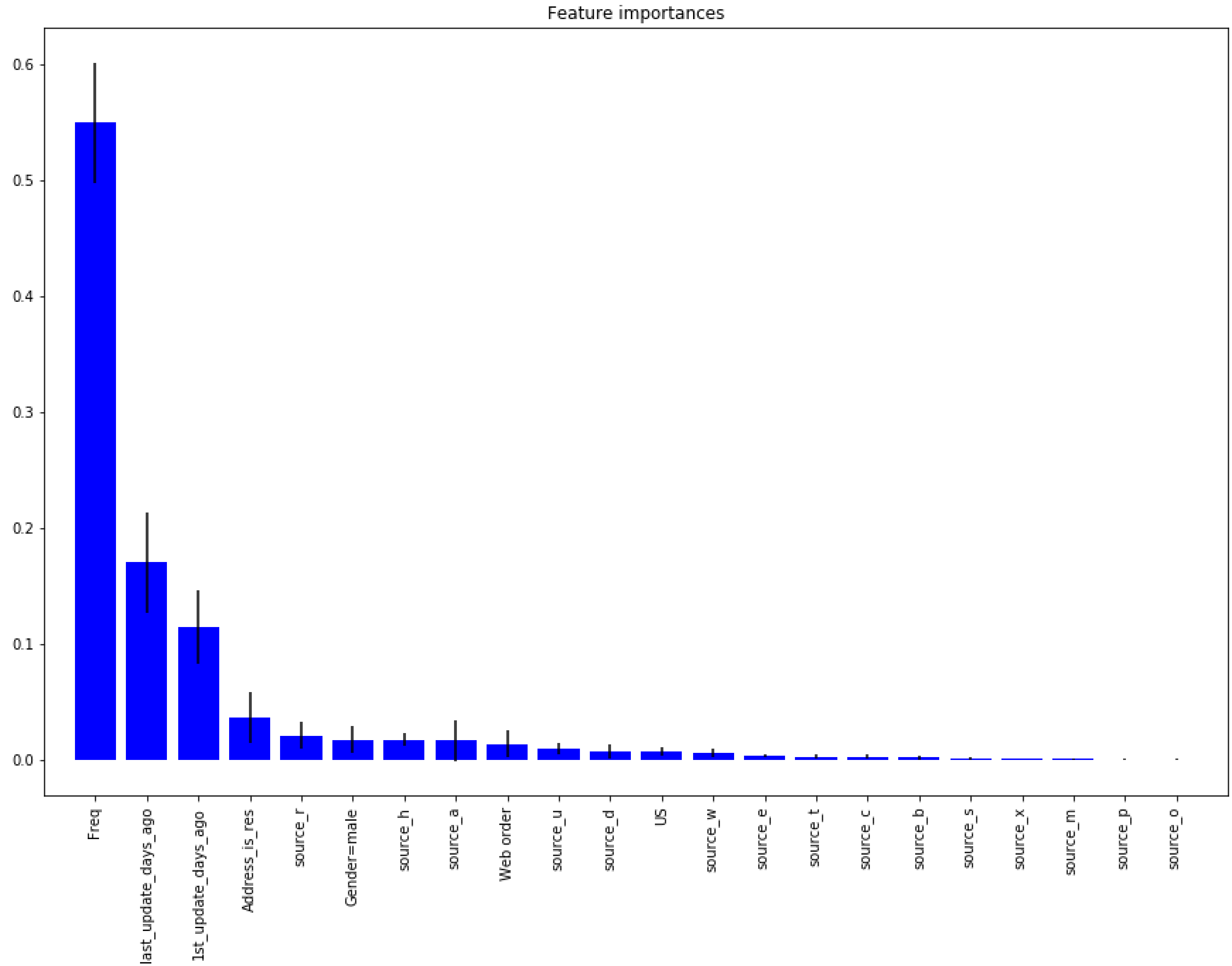
We only need three features at most, based on this principal component analysis.

Now let's see which features are the ones that actually explain the most variance.

In [14]:

```python
## SET A VARIABLE TO SWAP IN AND OUT BETWEEN THE DIFFERENT TRAINING INSTANCES

#Z = X_train_robust
#Z = X_train_norm
#Z = X_train_power
Z = X_train_standard

## Start with identifying the best features using a Random Forest classifier

## Create a new classifier
clf_rf_5 = ensemble.RandomForestRegressor()
clr_rf_5 = clf_rf_5.fit(Z, y_train)

## Save our importances to a variable
importances = clr_rf_5.feature_importances_

## Get the standard deviation for each feature
std = np.std([tree.feature_importances_ for tree in clf_rf_5.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

## Print the feature ranking
print("Feature ranking:")

## Print the top five features, and their importance based on the Random Forest Classifier
for f in range(0, 5):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))


## Plot the feature importances of the Random Forest Regressor - to see this visually

## Set the plot size
plt.figure(1, figsize=(15, 10))

## Set the title
plt.title("Feature importances")

## Plot a graph using all of the normalized features
plt.bar(range(Z.shape[1]), importances[indices],
       color="b", yerr=std[indices], align="center")

plt.xticks(range(Z.shape[1]), feature_cols[indices], rotation=90)
plt.xlim([-1, Z.shape[1]])

## Show the graph!
plt.show()
```

```
Feature ranking:
1. feature 16 (0.549566)
2. feature 17 (0.169993)
3. feature 18 (0.114418)
4. feature 21 (0.036042)
5. feature 9 (0.021158)
```



Confirming my earlier observation, only three of the features explain 82% of the variance between all the features.

In [15]:
```python
### Build a custom function to transform feature column or input into a scaled and then labeled item
def robustscale_labelencoder(array):

    ## Initialize a RobustScaler object
    rs = RobustScaler()
    ## Initialize a LabelEncoder object
    le = LabelEncoder()

    ## Reshape the array to conform to the right dimensions needed for RobustScaler
    array = array.reshape(-1, 1)

    ## Transform array using Robust Scaling
    robust_array = rs.fit_transform(array)

    ## Reshape the array back to its original shape for Label Encoding
    robust_array = robust_array.flatten()

    ## Now apply the LabelEncoding on a better scaled version of the data
    labeled_array = le.fit_transform(robust_array)

    ## Return the output
    return labeled_array
```

In [18]:
```python
## Use that cool function to transform the training target for use in the LR Classifier to observe what
## features are important
# y_train_robust_labeled = robustscale_labelencoder(y_train)

## Use StandardScaling on the training target.  Not likely to help with training the model, from what I have seen
## in previous attempts to train the models

y_train_standard = StandardScaler().fit_transform(y_train.reshape(-1, 1))

y_train_labeled = LabelEncoder().fit_transform(y_train)
```

In [20]:
```python
## Build RF classifier to use in feature selection
clf = linear_model.LogisticRegression(C=1e5)

# Build step forward feature selection
sfs1 = sfs(clf,
           k_features=6,
           forward=True, # Otherwise, this will be the backward selection
           floating=False,
           n_jobs=10, # The number of CPUs to use for evaluating
           verbose=2,
           scoring='accuracy',
           cv=5)

# Perform SFFS
sfs1 = sfs1.fit(X_train_standard, y_train_labeled)
```

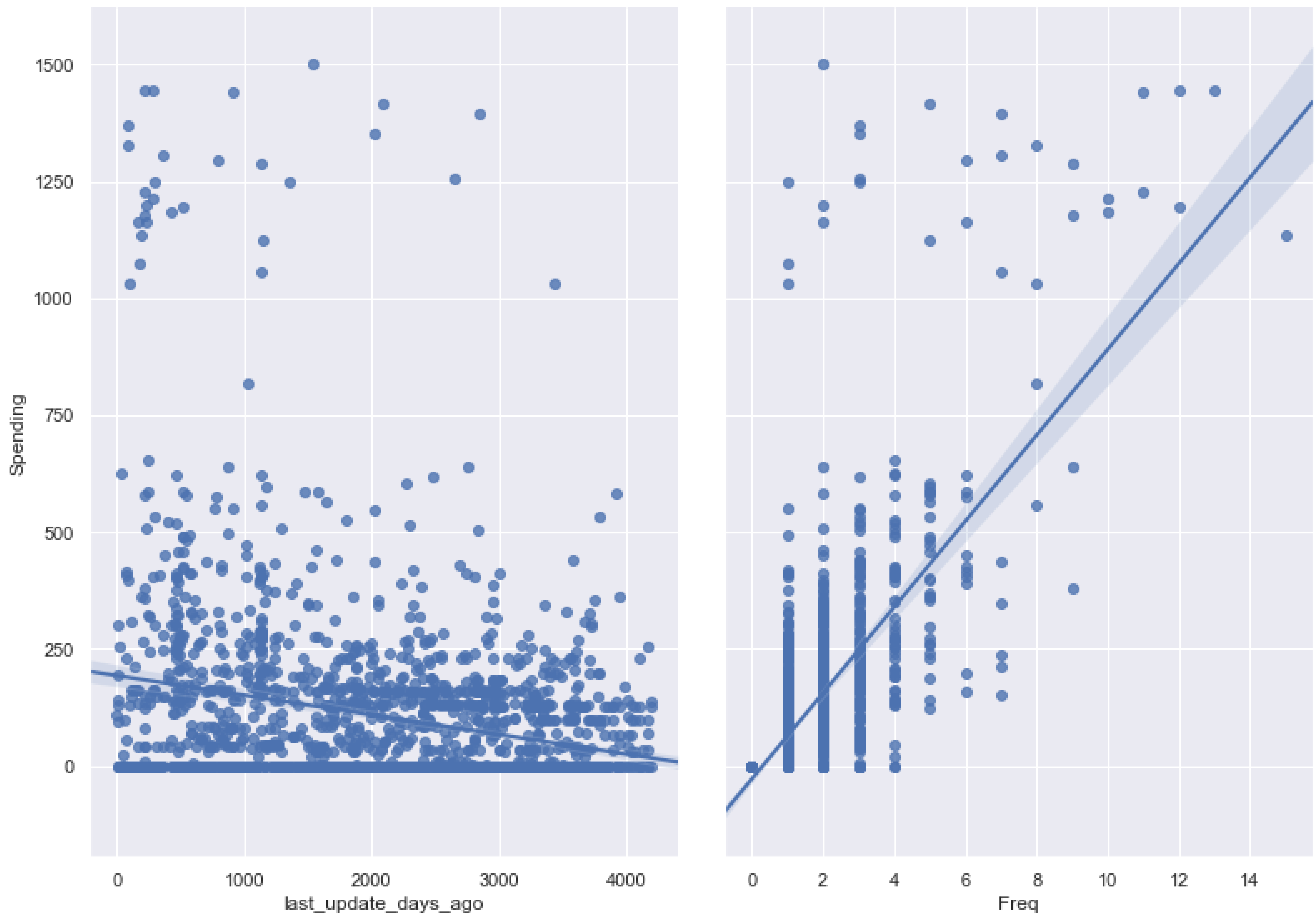
```
[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  15 out of  22 | elapsed:   54.2s remaining:   25.2s
[Parallel(n_jobs=10)]: Done  22 out of  22 | elapsed:   58.7s finished

[2019-10-30 14:17:06] Features: 1/6 -- score: 0.44553618151441493[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  13 out of  21 | elapsed:  1.2min remaining:   42.4s
[Parallel(n_jobs=10)]: Done  21 out of  21 | elapsed:  1.2min finished

[2019-10-30 14:18:22] Features: 2/6 -- score: 0.47563816778106094[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  12 out of  20 | elapsed:  1.4min remaining:   56.2s
[Parallel(n_jobs=10)]: Done  20 out of  20 | elapsed:  1.5min finished

[2019-10-30 14:19:50] Features: 3/6 -- score: 0.5165274559056995[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  10 out of  19 | elapsed:   59.4s remaining:   53.4s
[Parallel(n_jobs=10)]: Done  19 out of  19 | elapsed:  1.8min finished

[2019-10-30 14:21:39] Features: 4/6 -- score: 0.5331336778749486[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done   9 out of  18 | elapsed:  1.1min remaining:  1.1min
[Parallel(n_jobs=10)]: Done  18 out of  18 | elapsed:  2.0min finished

[2019-10-30 14:23:43] Features: 5/6 -- score: 0.5403199214808022[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done   7 out of  17 | elapsed:  1.3min remaining:  1.9min
[Parallel(n_jobs=10)]: Done  17 out of  17 | elapsed:  2.3min finished

[2019-10-30 14:26:03] Features: 6/6 -- score: 0.5471182950056331
```

In [21]:
```
## Now we print out what feature columns of Linear Regressor picked out - we can use these to subset later
## We'll remember to grab these columns further along in the analysis
feat_cols = list(sfs1.k_feature_idx_)
print(feat_cols)
```

```
[3, 7, 8, 16, 17, 21]
```

We do a little visual exploration of the features to see what the model is using for data points; this will help us with encoding the target correctly, as well as selecting the proper distance metric for kNN and the various forms of regression.

In [22]:
```
### Reset seaborn to the default background - for better viewing
sns.set()

## Build a quick plot comparing two of the features column with our spending using a simple regression line
sns.pairplot(purchases_df, x_vars=['last_update_days_ago', 'Freq'], y_vars='Spending', height = 8, aspect = 0.7, kind =
'reg')

## Show the plot
plt.show()
```

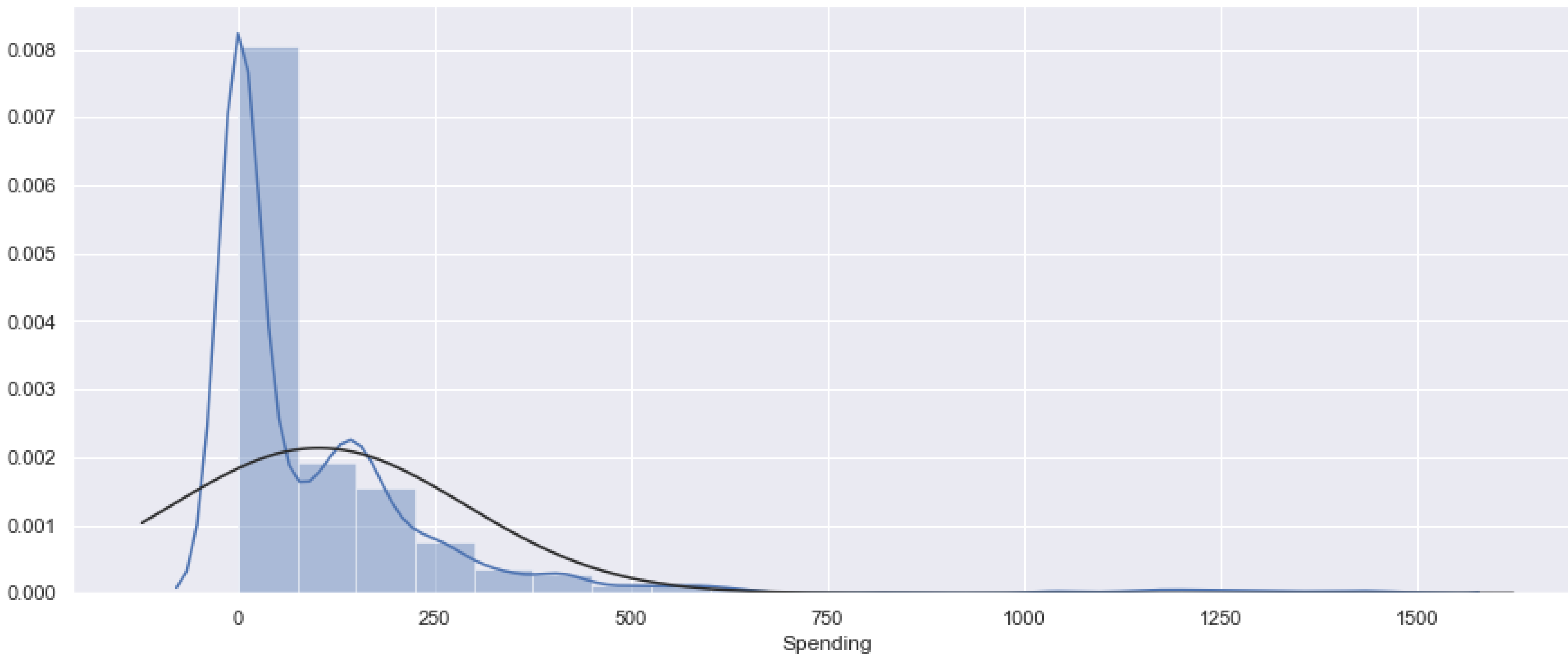


Interesting observation to be made here - the "last update" feature column has a tremendous amount of spread between how much a customer spent, and shows almost no linear relationship at all.

Conversely, the "freq" feature column is a little bit more linear, but not quite. So this is telling me that my linear model is probably going to suffer because it won't be able to distinguish between targets very well.

I will have to scale the features and targets correctly, or many of the data mining models will perform badly.

In [23]:
```python
## Plotting out the distribution of spending - to see how much a spread my target is

## Import a normal distribution from scipy
from scipy.stats import norm

## Build a new plot
plt.figure(1, figsize=(15, 6))

## Use a special function in seaborn to build a distribution plot and also include a
sns.distplot(purchases_df.Spending, fit = norm, kde = True, bins = 20)

## Show the plot
plt.show()
```



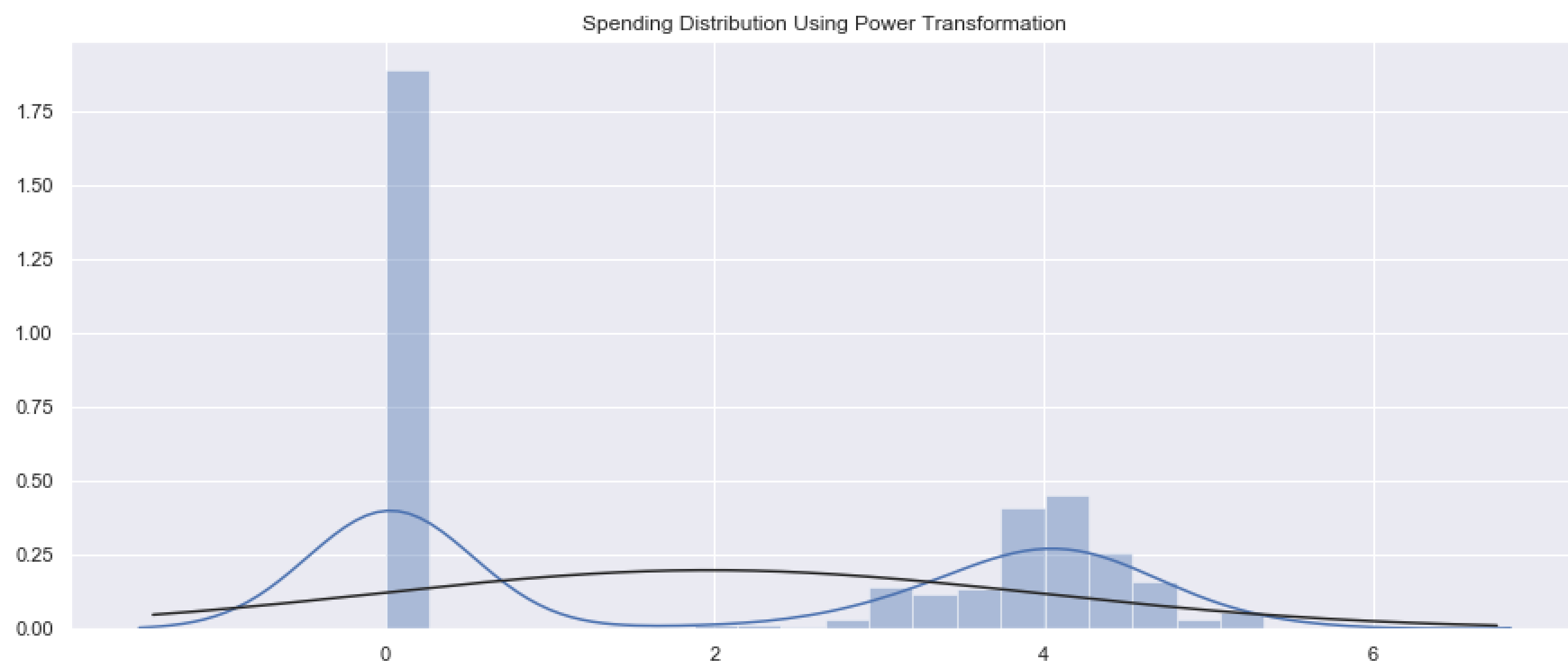
Most of the spending is less than $20! We will probably want to standardize/normalize the target first, and then use that as the prediction.

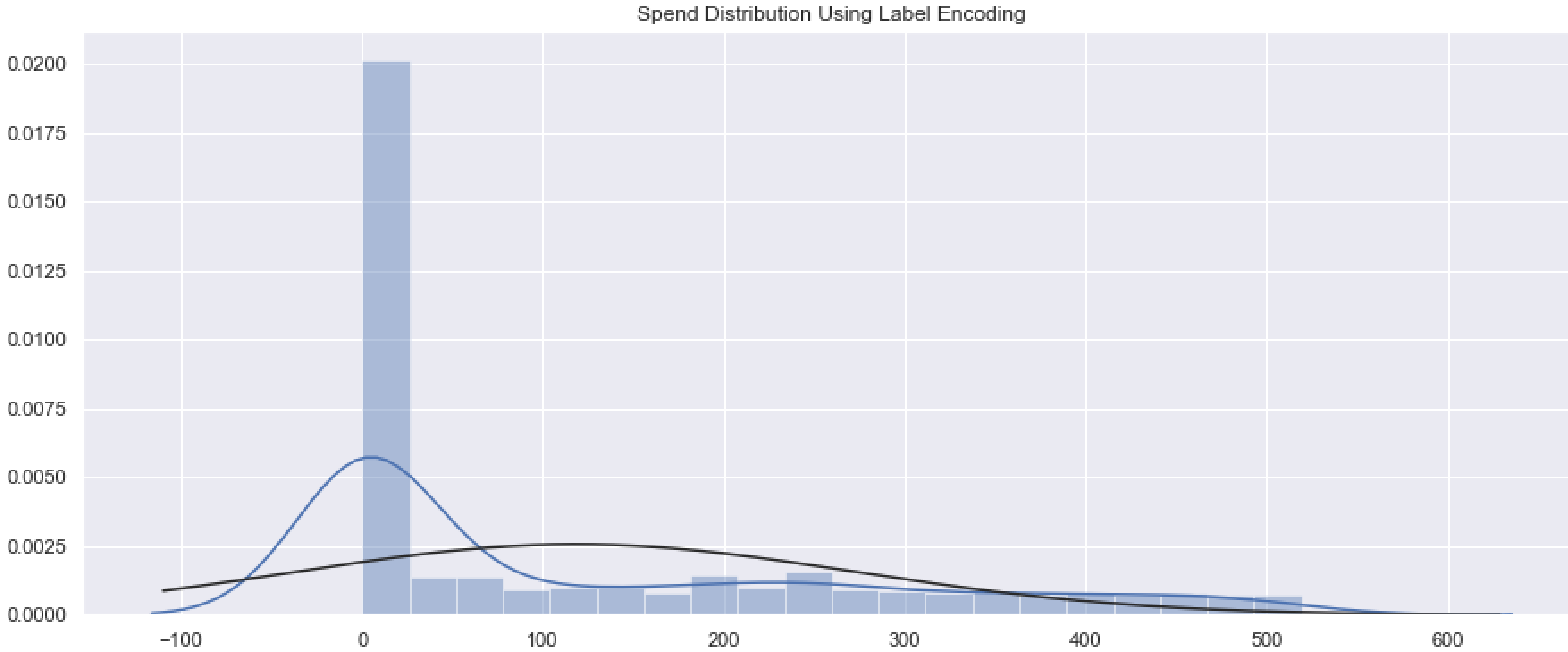
In [25]:

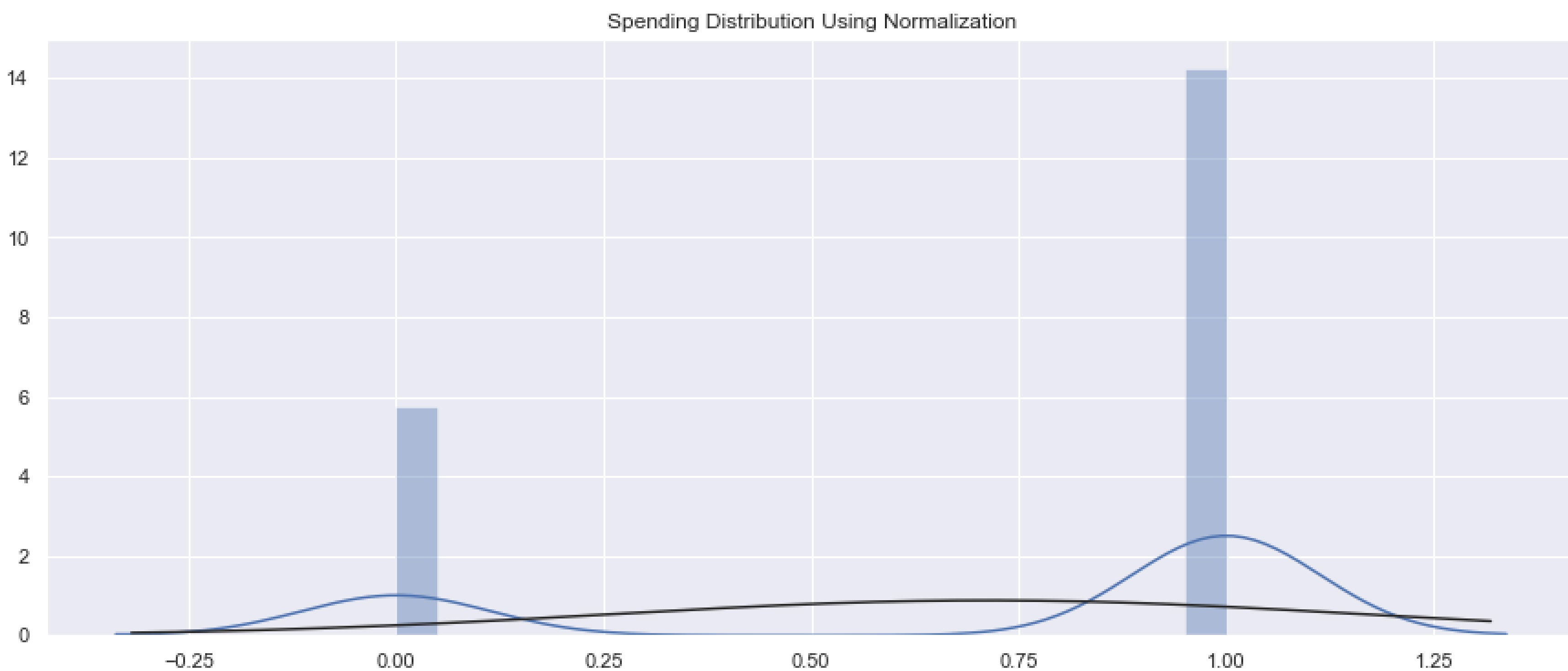
```
## Create two new encoders for the target - we'll use this to see
rs = RobustScaler()
pt = PowerTransformer(method='yeo-johnson', standardize=False)
le = LabelEncoder()
nm = Normalizer(norm = "l1")


## Reshape the data because these scaling methods are only supposed to be used by 2D arrays
y_train_reshape = y_train.reshape(-1, 1)

## Create four versions of the training data, to see the distributions for each one
robust_target = rs.fit_transform(y_train_reshape)
power_target = pt.fit_transform(y_train_reshape)
label_target = le.fit_transform(y_train)
normal_target = nm.fit_transform(y_train_reshape)
```

In [26]:
```
## Plotting out the distribution of spending using the new target variables with new encoding

target_list = [robust_target, power_target, label_target, normal_target]
titles = ["Spending Distribution Using Robust Scaling", "Spending Distribution Using Power Transformation",
          "Spend Distribution Using Label Encoding", "Spending Distribution Using Normalization"]

for i,t in zip(target_list, titles):

# Build a new plot
    plt.figure(1, figsize=(15, 6))

    ## Use a special function in seaborn to build a distribution plot and also include a
    sns.distplot(i, fit = norm, kde = True, bins = 20)

    plt.title(t)

    ## Show the plot
    plt.show()
```


Spending Distribution Using Robust Scaling
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
2
4
6
8
10


Spending Distribution Using Power Transformation
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
0
2
4
6


Spend Distribution Using Label Encoding
0.0200
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
−100
0
100
200
300
400
500
600


Spending Distribution Using Normalization
14
12
10
8
6
4
2
0
−0.25
0.00
0.25
0.50
0.75
1.00
1.25

Here's a analysis of the behavior of the different encoding strategies for the target variable. Based on the above distribution patterns, and knowing how the original "Spending" column is distributed, I will use normalize the target variable for training the model.

What this pre-processing work has told me is that

1. The target variable is heavily skewed to the left side, most purchases are going to be for low amounts of money
2. We will want to normalize the target variable, to get them on the same scale, otherwise the regression will suffer
3. We're dealing with a lot of outliers, especially the cases where the spend is quite large - we will want to use L1 normalization
4. Many of the features are binary variables and don't really account for the variance or spread
5. In fact, many of them are probably not needed. So we can build models using all the features or a subset

B. Model Creation and Evaluation

1. Create parameter grids for each model
2. Used nested cross validation to determine the best model
3. Tune the hyper parameters for the best model
4. Evaluate the models on the testing data

In [40]:

```python
### RE INITIALIZE OUR TARGETS AND THEN TRANSFORM THE LABELS AND TARGET FOR TRAINING

## Create our "X" variable that contains all the features we are curious about plotting
X = np.array(purchases_df[feature_cols])

## Create our "y" variable which is our target variable
y = np.array(purchases_df["Spending"])

## Split data training 70 % and testing 30%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)

## Normalize the features
X_train_normal = nm.fit_transform(X_train)

## Reshape y_train (target variable) for normalization technique
y_train_reshaped = y_train.reshape(-1, 1)
y_train_labeled = le.fit_transform(y_train)

## Print the results of the transformation
print(X_train_normal)
print()
print(y_train_labeled)
```

```
[[0.00024969 0.         0.00024969 ... 0.         0.00024969 0.        ]
 [0.00020665 0.         0.         ... 0.00020665 0.00020665 0.        ]
 [0.00030516 0.         0.         ... 0.00030516 0.         0.00030516]
 ...
 [0.00039573 0.         0.00039573 ... 0.00039573 0.         0.00039573]
 [0.         0.         0.         ... 0.         0.         0.        ]
 [0.         0.         0.         ... 0.00035625 0.         0.        ]]

[349 190   2 ...   0   0   0]
```

Here are the different models I will be using for my analysis of this regression problem

1. Linear Regression (requires all numeric)
2. k-Nearest Neighbors
3. Decision Tree (requires pre-processing)
4. SVM Regression (requires pre-processing of data)
5. Ensemble/Gradient Boost
6. Neural Network (Keras & KerasTuner)

In [36]:
```python
######### SETTING UP PARAMETER GRIDS FOR THE DIFFERENT DATA MINING MODELS WE ARE GOING TO USE ############

## Set up a grid for the Logit Regressor
## Using L1 only, this is a sparse data set.  And this data mining technique is going to suffer anyways
lr_p_grid = {"penalty": ["l1"],
            "C": [1, 5, 10, 50, 1000],
            "solver": ["liblinear"]}

## Set up a grid for kNN Regressor
## Going to use 1-30 neighbors, and two different distance calculations
knnr_p_grid = {"n_neighbors": list(range(1, 31)),
             "weights": ["uniform", "distance"]}

## Set up a grid for the DecisionTree Regressor
dtr_p_grid = {"criterion": ["mse", "mae"],
            "splitter": ["best", "random"],
            "max_features": [4, 6, 10, 15],
            "max_depth": [5, 10, 15]}

## Set up a grid for the Support Vector Regressor
svr_p_grid = {"C": [1, 5, 10, 50, 1000],
              "gamma": [0.0001, 0.0005, 0.001, 0.005],
             "kernel": ["poly", "rbf"]}

## Set up a grid for GBoost
### Use least squares for the regression
gbr_p_grid = {'loss': ['ls'],
                 'n_estimators': [100, 200, 300, 400, 500],
                 'max_depth': [3, 4, 5],
                 'min_samples_split': [2, 4, 6],
                 'learning_rate': [0.01]}
```

In [37]:
```python
## Set a number of trials to run for the models

num_trials = 20

## Empty arrays to store scores for classifier
nested_scores_lr = np.zeros(num_trials)
nested_scores_knnr = np.zeros(num_trials)
nested_scores_dtr = np.zeros(num_trials)
nested_scores_svr = np.zeros(num_trials)
nested_scores_gbr = np.zeros(num_trials)
```

In [38]:
```python
## Create new regressors for each data mining technique

## Linear Regression
lr = linear_model.LogisticRegression()

## k-Nearest Neighbors
knnr = neighbors.KNeighborsRegressor()

## Decision Tree
dtr = tree.DecisionTreeRegressor()

## Support Vector Machine
svr = svm.SVR()

## Gradient Boost
gbr = ensemble.GradientBoostingRegressor()
```

In [39]:
```python
## All of the new regressors initialized correctly
print(lr, knnr, dtr, svr, gbr)
```

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False) KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkows
ki',
          metric_params=None, n_jobs=None, n_neighbors=5, p=2,
          weights='uniform') DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=None, splitter='best') SVR(C=1.0, cache_size=200, coef0=0.0, degree=3, epsilon=
0.1,
  gamma='auto_deprecated', kernel='rbf', max_iter=-1, shrinking=True,
  tol=0.001, verbose=False) GradientBoostingRegressor(alpha=0.9, criterion='friedman_mse', init=None,
             learning_rate=0.1, loss='ls', max_depth=3, max_features=None,
             max_leaf_nodes=None, min_impurity_decrease=0.0,
             min_impurity_split=None, min_samples_leaf=1,
             min_samples_split=2, min_weight_fraction_leaf=0.0,
             n_estimators=100, n_iter_no_change=None, presort='auto',
             random_state=None, subsample=1.0, tol=0.0001,
             validation_fraction=0.1, verbose=0, warm_start=False)
```

Now we'll use Nested Cross Validation, combined with `GridSearch` to find out what models perform well.

I am going to build one loop for each regression method to reduce the load on my machine

In [52]:
```
####### Training ground - just to make sure the data is being used well

dtr.fit(X_train_normal, y_train_labeled)
```

Out[52]:
```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_features=None,
           max_leaf_nodes=None, min_impurity_decrease=0.0,
           min_impurity_split=None, min_samples_leaf=1,
           min_samples_split=2, min_weight_fraction_leaf=0.0,
           presort=False, random_state=None, splitter='best')
```

In [59]:
```
## We get a lot of convergence warnings, because the data mining procedures get lost
## I just include this to remove the warnings.

import warnings
warnings.filterwarnings(action = "ignore", module = "sklearn")
```

In [60]:

```python
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    lreg= GridSearchCV(estimator=lr, param_grid=lr_p_grid, cv=inner_cv)
    lreg.fit(X_train_normal, y_train_labeled)

    nested_score = cross_val_score(lreg, X = X_train_normal, y = y_train_labeled, cv = outer_cv)
    nested_scores_lr[i] = nested_score.mean()
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-60-b4a3015e8336> in <module>
     12     lreg.fit(X_train_normal, y_train_labeled)
     13 
---> 14     nested_score = cross_val_score(lreg, X = X_train_normal, y = y_train_labeled, cv = outer_cv)
     15     nested_scores_lr[i] = nested_score.mean()

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in cross_val_score(estimator, X, y, groups, scorin
g, cv, n_jobs, verbose, fit_params, pre_dispatch, error_score)
    400                                 fit_params=fit_params,
    401                                 pre_dispatch=pre_dispatch,
--> 402                                 error_score=error_score)
    403     return cv_results['test_score']
    404 

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in cross_validate(estimator, X, y, groups, scoring,
cv, n_jobs, verbose, fit_params, pre_dispatch, return_train_score, return_estimator, error_score)
    238             return_times=True, return_estimator=return_estimator,
    239             error_score=error_score)
--> 240         for train, test in cv.split(X, y, groups))
    241 
    242     zipped_scores = list(zip(*scores))

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    918                 self._iterating = self._original_iterator is not None
    919 
--> 920             while self.dispatch_one_batch(iterator):
    921                 pass
    922 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
```

```
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550 
    551     def get(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _fit_and_score(estimator, X, y, scorer, train, t
est, verbose, parameters, fit_params, return_train_score, return_parameters, return_n_test_samples, return_times, retu
rn_estimator, error_score)
    526             estimator.fit(X_train, **fit_params)
    527         else:
--> 528             estimator.fit(X_train, y_train, **fit_params)
    529 
    530     except Exception as e:

C:\Python\lib\site-packages\sklearn\model_selection\_search.py in fit(self, X, y, groups, **fit_params)
    720                 return results_container[0]
    721 
--> 722             self._run_search(evaluate_candidates)
    723 
    724         results = results_container[0]

C:\Python\lib\site-packages\sklearn\model_selection\_search.py in _run_search(self, evaluate_candidates)
   1189     def _run_search(self, evaluate_candidates):
   1190         """Search all candidates in param_grid"""
-> 1191         evaluate_candidates(ParameterGrid(self.param_grid))
   1192 
   1193 
```

```
C:\Python\lib\site-packages\sklearn\model_selection\_search.py in evaluate_candidates(candidate_params)
    709                                for parameters, (train, test)
    710                                in product(candidate_params,
--> 711                                           cv.split(X, y, groups)))
    712 
    713                 all_candidate_params.extend(candidate_params)

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    918                 self._iterating = self._original_iterator is not None
    919 
--> 920             while self.dispatch_one_batch(iterator):
    921                 pass
    922 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550 
    551     def get(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
```

```
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _fit_and_score(estimator, X, y, scorer, train, t
est, verbose, parameters, fit_params, return_train_score, return_parameters, return_n_test_samples, return_times, retu
rn_estimator, error_score)
    526             estimator.fit(X_train, **fit_params)
    527         else:
--> 528             estimator.fit(X_train, y_train, **fit_params)
    529 
    530     except Exception as e:

C:\Python\lib\site-packages\sklearn\linear_model\logistic.py in fit(self, X, y, sample_weight)
   1303                 self.class_weight, self.penalty, self.dual, self.verbose,
   1304                 self.max_iter, self.tol, self.random_state,
-> 1305                 sample_weight=sample_weight)
   1306             self.n_iter_ = np.array([n_iter_])
   1307             return self

C:\Python\lib\site-packages\sklearn\svm\base.py in _fit_liblinear(X, y, C, fit_intercept, intercept_scaling, class_wei
ght, penalty, dual, verbose, max_iter, tol, random_state, multi_class, loss, epsilon, sample_weight)
    921         X, y_ind, sp.isspmatrix(X), solver_type, tol, bias, C,
    922         class_weight_, max_iter, rnd.randint(np.iinfo('i').max),
--> 923         epsilon, sample_weight)
    924     # Regarding rnd.randint(..) in the above signature:
    925     # seed for srand in range [0..INT_MAX); due to limitations in Numpy

KeyboardInterrupt: 
```

In [62]:
```
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    knnreg= GridSearchCV(estimator=knnr, param_grid=knnr_p_grid, cv=inner_cv)
    knnreg.fit(X_train_normal, y_train_labeled)

    nested_score = cross_val_score(knnreg, X = X_train_normal, y = y_train_labeled, cv = outer_cv)
    nested_scores_knnr[i] = nested_score.mean()
```

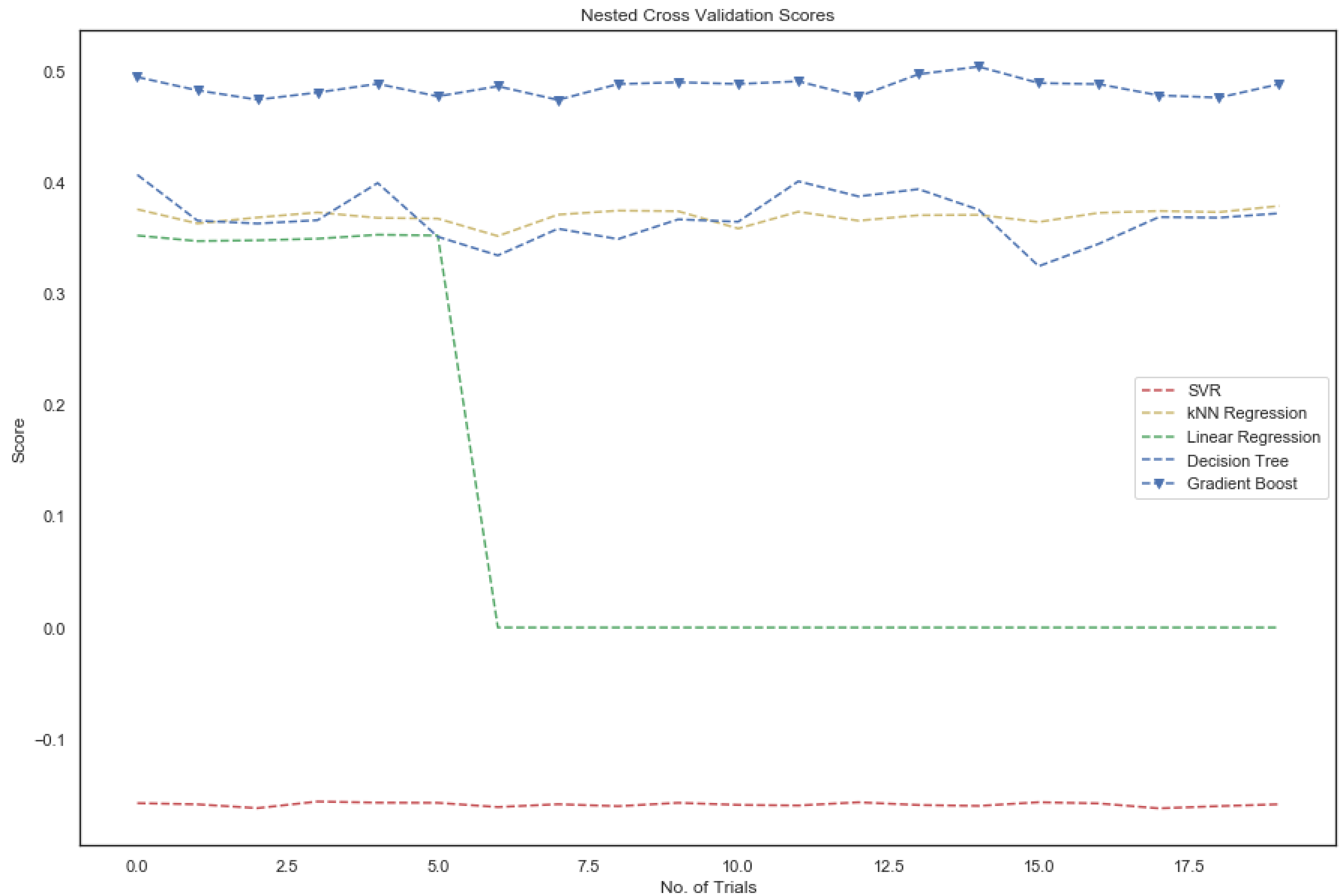
In [64]:
```
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    dtreg= GridSearchCV(estimator=dtr, param_grid=dtr_p_grid, cv=inner_cv)
    dtreg.fit(X_train_normal, y_train_labeled)

    nested_score = cross_val_score(dtreg, X = X_train_normal, y = y_train_labeled, cv = outer_cv)
    nested_scores_dtr[i] = nested_score.mean()
```

In [67]:
```python
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logistic Regression

    svreg= GridSearchCV(estimator=svr, param_grid=svr_p_grid, cv=inner_cv)
    svreg.fit(X_train_normal, y_train_labeled)

    nested_score = cross_val_score(svreg, X = X_train_normal, y = y_train_labeled, cv = outer_cv)
    nested_scores_svr[i] = nested_score.mean()
```

In [70]:
```python
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    gbreg= GridSearchCV(estimator=gbr, param_grid=gbr_p_grid, cv=inner_cv)
    gbreg.fit(X_train_normal, y_train_labeled)

    nested_score = cross_val_score(gbreg, X = X_train_normal, y = y_train_labeled, cv = outer_cv)
    nested_scores_gbr[i] = nested_score.mean()
```

In [263]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot nested scores for each classifier - quickly visual the best performing model
## This is WITHOUT having changed any of the default parameters
plt.plot(nested_scores_svr, 'r--', label = "SVR")
plt.plot(nested_scores_knnr, 'y--', label = "kNN Regression")
plt.plot(nested_scores_lr, 'g--', label = "Linear Regression")
plt.plot(nested_scores_dtr, 'b--', label = "Decision Tree")
plt.plot(nested_scores_gbr, 'v--', label = "Gradient Boost")


## Give some labels
plt.xlabel("No. of Trials")
plt.ylabel("Score")

## Title and Legend
plt.title("Nested Cross Validation Scores")
plt.legend(loc = 'center right')

## Show the graph
plt.show()
```



What this plot tells us is our models really suffer and don't do well with multivariate regression, which is what this problem is. We're going to try using a different method, namely Keras & Tensorflow to build a Neural Network.

Here we start building a neural network; it allows us to build from the ground up, continually learning, and will hopefully lead to much better performance.

In [18]:
```
## Import our libraries

from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout
from keras.utils import np_utils
from tensorflow.keras.callbacks import EarlyStopping
from tensorflow.keras.optimizers import SGD
```

```
Using TensorFlow backend.
```

In [9]:
```
## Remove any columns that will not be used as features
## I am making a conscious decision to remove sequence_number - I believe it is extraneous and not needed
## It won't help the predictive power of the models

feature_cols = purchases_df.columns[~purchases_df.columns.isin(["sequence_number", "Purchase", "Spending"])]
feature_cols
```

Out[9]:
```
Index(['US', 'source_a', 'source_c', 'source_b', 'source_d', 'source_e',
       'source_m', 'source_o', 'source_h', 'source_r', 'source_s', 'source_t',
       'source_u', 'source_p', 'source_x', 'source_w', 'Freq',
       'last_update_days_ago', '1st_update_days_ago', 'Web order',
       'Gender=male', 'Address_is_res'],
      dtype='object')
```

In [10]:
```
## Create a new "X" variable that contains all the features we are curious about plotting
X = np.array(purchases_df[feature_cols])

## Create our "y" variable which is our target variable and remove the floating point decimals
y = np.array(purchases_df["Spending"])

## Split data training 70 % and testing 30%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

In [11]:
```
## Standardize dataset
## Create new scaling object

sc = StandardScaler()

## Standardize the training data
X_train_standard = sc.fit_transform(X_train)
y_train_standard = sc.fit_transform(y_train.reshape(len(y_train),1))[:,0]
```

In [21]:
```
## Build a new model using Keras, with 22 feature columns

## Rectified Linear Unit - f(x) = max_value for x >= max_value, f(x) = x for threshold <= x < max_value, f(x) = alpha * (x - threshold)
## Initializer - he_uniform - It draws samples from a uniform distribution within [-limit, limit] where limit is sqrt(6 / fan_in) where fan_in is the number of input units in the weight tensor.
model = Sequential()
model.add(Dense(25, input_dim=22, activation='relu', kernel_initializer='he_uniform'))

## Activation - Linear to perform linear regression
model.add(Dense(1, activation='linear'))
```

In [23]:
```
## Standardize the testing data separately from the training data

X_test_standard = sc.transform(X_test)
y_test_standard = sc.transform(y_test.reshape(len(y_test),1))[:,0]
```

In [24]:
```
## Build the optimizer with stochastic gradient descent with a learning rate of 0.01 and a momentum of 0.9

opt = SGD(lr=0.01, momentum=0.9)
model.compile(loss="mean_squared_logarithmic_error", optimizer=opt, metrics = ["mse"])

## fit model
history = model.fit(X_train_standard, y_train_standard,
            validation_data=(X_test_standard, y_test_standard), epochs=100, verbose=0)
```

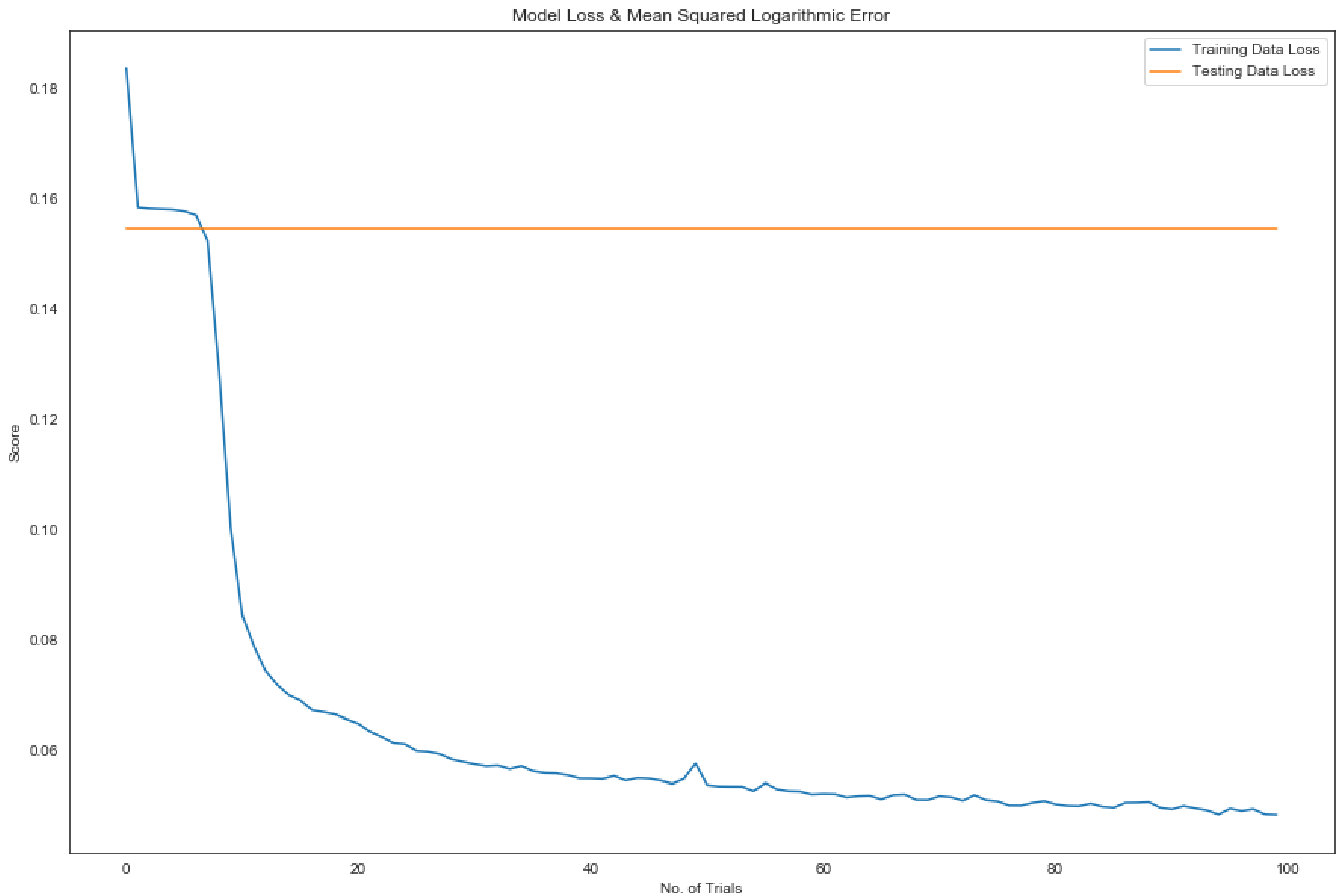
In [25]:
```
## Evaluate the model once it has been fit to determine performance

train_mse = model.evaluate(X_train_standard, y_train_standard, verbose=0)
test_mse = model.evaluate(X_test_standard, y_test_standard, verbose=0)
```

The Mean Squared Error, or MSE, loss is the default loss function to use for most regression problems, and this would normally be the logical choice. It is the preferred loss function under the inference framework of maximum likelihood if the distribution of the target is a Gaussian (normal) distrubtion.

In this case, it is a regression problem where the target values have a large spread of values, from $0 to 1500, and when predicting a larger value, we do not want to punish the model as heavily as with MSE.

Instead, we can calculate the natural logarithm of each of the predicted values, then calculate the mean squared error. This is called the Mean Squared Logarithmic Error loss, or MSLE for short.

In [26]:
```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot nested scores for each classifier - quickly visual the best performing model
## This is WITHOUT having changed any of the default parameters
plt.plot(history.history['loss'], label = "Training Data Loss")
plt.plot(history.history['val_loss'], label = "Testing Data Loss")

## Give some labels and title
plt.xlabel("No. of Epochs")
plt.ylabel("Score")

## Title and Legend
plt.title("Model Loss & Mean Squared Logarithmic Error")
plt.legend()

## Show the graph
plt.show()
```


Model Loss & Mean Squared Logarithmic Error
Training Data Loss
Testing Data Loss
Score
0.18
0.16
0.14
0.12
0.10
0.08
0.06
0
20
40
60
80
100
No. of Trials

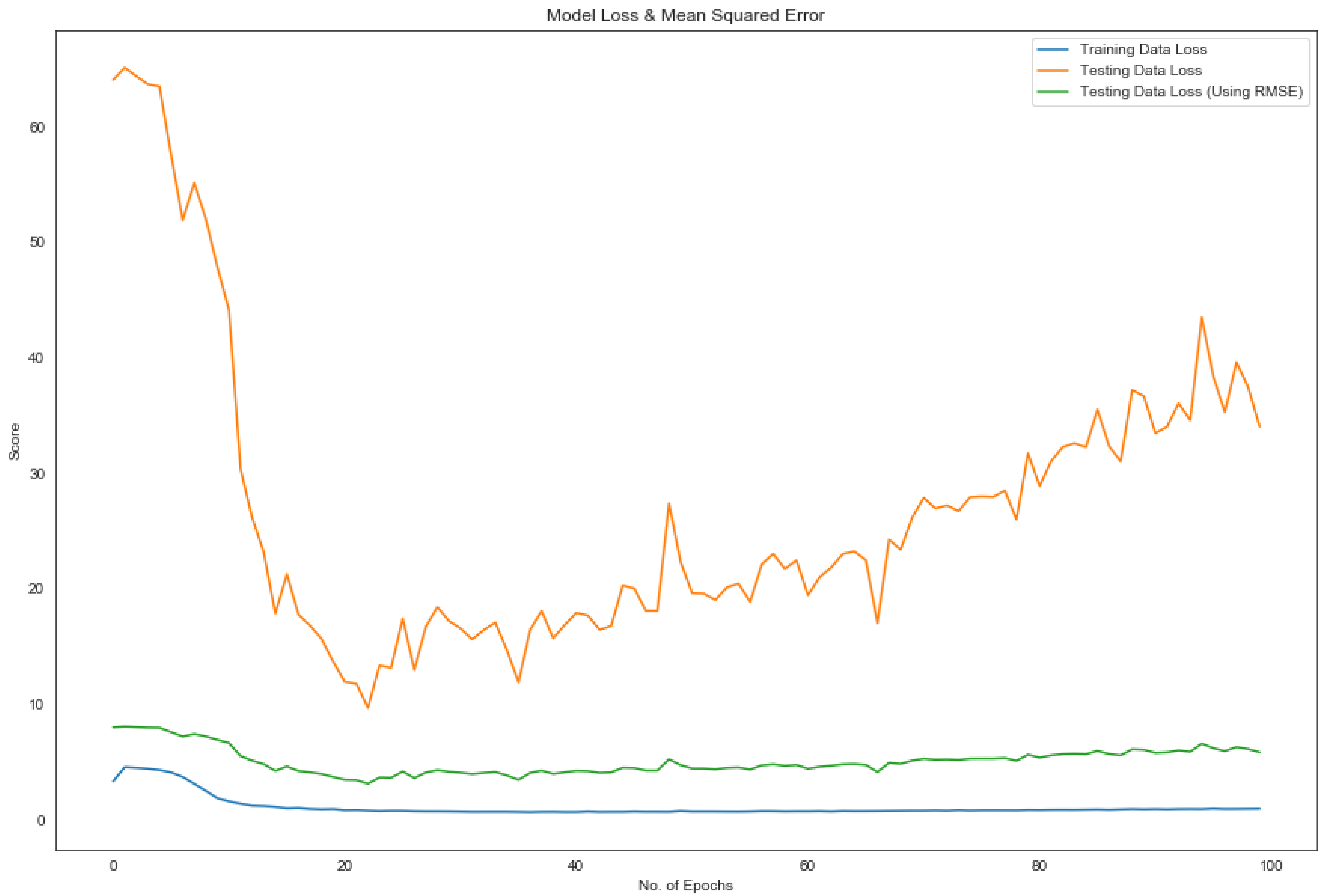
I created a line plot to show the Mean Squared Logarithmic Error (MSLE) loss over the training cycles on both the training data (blue line) and testing data (orange line).

We can see that the model learned reasonably quickly (after about 15 epochs) and test performance remained equivalent/steady from the moment the model was created because of the loss function being used.

The performance and convergence behavior of the model suggests that MSLE is a good match for a neural network model attempting to learn from this problem.

I will also want to see the Mean Squared Error (MSE) performance of the model, because the logarithmic loss function puts all of our "Purchases" on an expontential scale (the power of 10). As discovered earlier, most of the purchases are for less than $20.00, so we have to make sure the model can handle identifying smaller amounts.

In [28]:
```
from math import sqrt

### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot nested scores for each classifier - quickly visual the best performing model
## This is WITHOUT having changed any of the default parameters
plt.plot(history.history['mse'], label = "Training Data Loss")
plt.plot(history.history['val_mse'], label = "Testing Data Loss")

## Add more line for plotting the RMSE of testing data
plt.plot([sqrt(i) for i in history.history['val_mse']], label = "Testing Data Loss (Using RMSE)")

## Give some labels and title
plt.xlabel("No. of Epochs")
plt.ylabel("Score")

## Title and Legend
plt.title("Model Loss & Mean Squared Error")
plt.legend()

## Show the graph
plt.show()
```


Model Loss & Mean Squared Error
Training Data Loss
Testing Data Loss
Testing Data Loss (Using RMSE)
Score
No. of Epochs
0
10
20
30
40
50
60
0
20
40
60
80
100

This plot shows the Mean Squared Error loss over the training cycles for training data (blue line) and testing data (orange line); I also created a line to show the Root Mean Squared Error (RMSE) for testing data to really show the model performance.

It appears that MSE may be showing signs of over-fitting the problem. When I look at the RMSE, it is much more stable and doesn't fluctuate quite as much when going through training cycles. While I am confident that this model is performing much better compared to my earlier attempts with the "out-of-box" regressors from `sklearn`, I will run a similar process on the data but remove any rows that did not have a purchase to evaluate the model performance using this method.

(b) (20 points) As a variation on this exercise, create a separate "restricted" dataset (i.e., a subset of the original dataset), which includes only purchase records (i.e., where Purchase = 1). Build numeric prediction models to predict Spending for this restricted dataset. All the same requirements as for task (a) apply.

In [29]:

```
## Create a subset of the data, so that we only try to predict the Spending - cool!

## Subset the dataframe created earlier to only include rows with Spending information
purchase_only_df = purchases_df[purchases_df.Purchase != 0]
```

In [30]:

```
## Look at the first five observations

purchase_only_df.head()
```

Out[30]:

	sequence_number	US	source_a	source_c	source_b	source_d	source_e	source_m	source_o	source_h	...	source_x	source_w	Freq	las
0	1	1	0	0	1	0	0	0	0	0	...	0	0	2	
2	3	1	0	0	0	0	0	0	0	0	...	0	0	2	
8	9	1	1	0	0	0	0	0	0	0	...	0	0	4	
9	10	1	1	0	0	0	0	0	0	0	...	0	0	1	
13	14	1	1	0	0	0	0	0	0	0	...	0	0	5	

5 rows × 25 columns

In [31]:
```
## Confirm that no records without a purchase made it in

purchase_only_df.groupby("Purchase")["US"].count()
```

Out[31]:
```
Purchase
1    1000
Name: US, dtype: int64
```

In [33]:
```
## Create a new "X" variable that contains all the features
X = np.array(purchase_only_df[feature_cols])

## Create our "y" variable which is our target variable and remove the floating point decimals
y = np.array(purchase_only_df["Spending"])

## Split data training 70 % and testing 30%
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42)
```

In [34]:
```
## Standardize training features and target variable

sc = StandardScaler()

X_train_standard = sc.fit_transform(X_train)
y_train_standard = sc.fit_transform(y_train.reshape(len(y_train),1))[:,0]
```

In [35]:
```
## Standardize testing features and target variable separately so that the training data doesn't skew or affect the mod
el performance
## This is known as DATA LEAKAGE.

X_test_standard = sc.transform(X_test)
y_test_standard = sc.transform(y_test.reshape(len(y_test),1))[:,0]
```

C. Use helpful Keras & Keras Tuner libraries to help us perform model evaluation at the same time we develop new classifiers

1. Build the model function
2. Build the hyper tuner function
3. Tune the model and explore the data space to make predictions
4. Assess the model performance.

We can use some specific custom functions which are explained below.

In [ ]:
```python
### Import some libraries to help us build and tune the model at the same time

from tensorflow import keras
from kerastuner.tuners import RandomSearch
from tensorflow.keras import layers
```

In [45]:
```python
## As a first step, we build a function to actually put together our model

def build_model(hp):

    ## Base layer
    model = Sequential()

    ## Add first layer, test with 10 up to 22 features
    model.add(layers.Dense(units = hp.Int ("units",
                                          min_value = 10,
                                          max_value = 22,
                                          step = 2),
                          ## Use same initializer as model above
                          activation = "relu", kernel_initializer = "he_uniform"))

    ## Add target layer using linear regressor
    model.add(layers.Dense(1, activation = "linear"))

    ## Use different learning rates to test the model
    model.compile(
    optimizer = keras.optimizers.SGD(
        hp.Choice("learning_rate", values = [1e-2, 1e-3, 1e-4])),


    ## Use same loss fucntion as before
    loss = "mean_squared_logarithmic_error",

    ## Test using MSE

    metrics = ["mse"])

    ## Return completed model
    return model
```

In [46]:
```
## Second step, we build a "hypertuner", this is what will actually fine tune our model as it is being created

### Use RandomSearch

tuner = RandomSearch(

    ## Use model function from above
    build_model,

    ## What is objective function?  Using loss here
    objective = "val_loss",

    ## Set number of trials
    max_trials = 5,

    ## Number of executions
    executions_per_trial = 3,

    ## Set to short dir path
    directory = "C:\\"

)
```

In [47]:

```
## Run our search with Keras Tuner!

tuner.search(X_train_standard, y_train_standard, epochs = 10,
             validation_data = (X_test_standard, y_test_standard))
```

```
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 9s - loss: 0.2278 - mse: 2.077 - ETA: 0s - loss: 0.1906 - mse: 1.850 -
1s 1ms/sample - loss: 0.1897 - mse: 1.8145 - val_loss: 0.2142 - val_mse: 39.1103
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.3552 - mse: 3.114 - 0s 113us/sample - loss: 0.1753 - mse:
1.8438 - val_loss: 0.2142 - val_mse: 44.3810
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1800 - mse: 1.620 - 0s 125us/sample - loss: 0.1688 - mse:
1.8697 - val_loss: 0.2142 - val_mse: 47.8374
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.0647 - mse: 1.080 - ETA: 0s - loss: 0.1623 - mse: 1.822 -
0s 140us/sample - loss: 0.1655 - mse: 1.9067 - val_loss: 0.2142 - val_mse: 49.9367
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1368 - mse: 2.110 - ETA: 0s - loss: 0.1617 - mse: 1.921 -
0s 140us/sample - loss: 0.1632 - mse: 1.9334 - val_loss: 0.2142 - val_mse: 51.1677
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.1585 - mse: 1.573 - 0s 125us/sample - loss: 0.1612 - mse:
1.8967 - val_loss: 0.2142 - val_mse: 52.5365
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.1937 - mse: 1.670 - ETA: 0s - loss: 0.1383 - mse: 1.852 -
0s 145us/sample - loss: 0.1596 - mse: 1.9086 - val_loss: 0.2142 - val_mse: 53.2560
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.1789 - mse: 1.591 - ETA: 0s - loss: 0.1681 - mse: 1.969 -
0s 136us/sample - loss: 0.1580 - mse: 1.8965 - val_loss: 0.2142 - val_mse: 53.7244
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1993 - mse: 1.889 - ETA: 0s - loss: 0.1622 - mse: 1.918 -
0s 139us/sample - loss: 0.1564 - mse: 1.8611 - val_loss: 0.2142 - val_mse: 53.9843
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.1509 - mse: 1.541 - ETA: 0s - loss: 0.1598 - mse: 1.886 -
0s 134us/sample - loss: 0.1547 - mse: 1.8316 - val_loss: 0.2142 - val_mse: 54.2806
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 12s - loss: 0.3805 - mse: 2.29 - 1s 1ms/sample - loss: 0.4098 - mse:
2.6610 - val_loss: 0.2142 - val_mse: 8.6957
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2388 - mse: 1.723 - ETA: 0s - loss: 0.2865 - mse: 2.146 -
0s 135us/sample - loss: 0.2780 - mse: 2.0689 - val_loss: 0.2142 - val_mse: 14.1440
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.2782 - mse: 2.654 - ETA: 0s - loss: 0.1944 - mse: 1.804 -
0s 183us/sample - loss: 0.2025 - mse: 1.9278 - val_loss: 0.2142 - val_mse: 18.3506
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.1523 - mse: 1.465 - ETA: 0s - loss: 0.1792 - mse: 2.003 -
0s 139us/sample - loss: 0.1713 - mse: 1.9377 - val_loss: 0.2142 - val_mse: 20.9990
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.0361 - mse: 0.891 - 0s 107us/sample - loss: 0.1598 - mse:
1.9953 - val_loss: 0.2142 - val_mse: 22.6407
```

```
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.2181 - mse: 2.926 - 0s 131us/sample - loss: 0.1556 - mse:
2.0410 - val_loss: 0.2142 - val_mse: 23.6571
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.2695 - mse: 4.305 - ETA: 0s - loss: 0.1439 - mse: 1.961 -
0s 152us/sample - loss: 0.1539 - mse: 2.0656 - val_loss: 0.2142 - val_mse: 24.5813
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.0666 - mse: 1.752 - ETA: 0s - loss: 0.1495 - mse: 2.102 -
0s 141us/sample - loss: 0.1524 - mse: 2.0858 - val_loss: 0.2142 - val_mse: 25.8277
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.0622 - mse: 1.494 - 0s 107us/sample - loss: 0.1515 - mse:
2.1006 - val_loss: 0.2142 - val_mse: 26.6425
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.2446 - mse: 3.798 - 0s 124us/sample - loss: 0.1509 - mse:
2.1524 - val_loss: 0.2142 - val_mse: 27.1363
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 14s - loss: 0.0780 - mse: 2.32 - 1s 1ms/sample - loss: 0.2018 - mse:
2.5607 - val_loss: 1.1964 - val_mse: 8.1160
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.1176 - mse: 2.172 - ETA: 0s - loss: 0.1897 - mse: 2.715 -
0s 162us/sample - loss: 0.1793 - mse: 2.6575 - val_loss: 0.8969 - val_mse: 5.4383
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.0769 - mse: 0.888 - ETA: 0s - loss: 0.1604 - mse: 2.538 -
0s 143us/sample - loss: 0.1674 - mse: 2.7262 - val_loss: 0.6836 - val_mse: 3.9247
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2739 - mse: 3.457 - ETA: 0s - loss: 0.1662 - mse: 2.968 -
0s 158us/sample - loss: 0.1604 - mse: 2.8242 - val_loss: 0.5364 - val_mse: 3.0590
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1883 - mse: 2.779 - ETA: 0s - loss: 0.1427 - mse: 2.591 -
0s 171us/sample - loss: 0.1560 - mse: 2.8656 - val_loss: 0.4387 - val_mse: 2.5670
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.0993 - mse: 2.312 - ETA: 0s - loss: 0.1529 - mse: 2.974 -
0s 325us/sample - loss: 0.1531 - mse: 2.9327 - val_loss: 0.3717 - val_mse: 2.2779
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.1290 - mse: 2.855 - ETA: 0s - loss: 0.1408 - mse: 2.685 -
0s 153us/sample - loss: 0.1509 - mse: 2.9367 - val_loss: 0.3217 - val_mse: 2.0908
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.0923 - mse: 1.974 - ETA: 0s - loss: 0.1468 - mse: 2.868 -
0s 134us/sample - loss: 0.1492 - mse: 2.9831 - val_loss: 0.2911 - val_mse: 1.9873
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1389 - mse: 3.140 - ETA: 0s - loss: 0.1449 - mse: 2.891 -
0s 182us/sample - loss: 0.1473 - mse: 2.9709 - val_loss: 0.2718 - val_mse: 1.9185
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.0388 - mse: 1.840 - ETA: 0s - loss: 0.1420 - mse: 2.870 -
0s 140us/sample - loss: 0.1456 - mse: 2.9571 - val_loss: 0.2577 - val_mse: 1.8692
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.01

|-units: 18

|-Score: 0.22868611564238864

|-Best step: 0

```
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 8s - loss: 0.4462 - mse: 3.820 - 1s 903us/sample - loss: 0.3916 - mse: 2.8532 - val_loss: 0.2239 - val_mse: 5.1755
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2346 - mse: 2.658 - ETA: 0s - loss: 0.2996 - mse: 2.604 - 0s 281us/sample - loss: 0.2902 - mse: 2.4810 - val_loss: 0.2142 - val_mse: 7.8681
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1844 - mse: 1.773 - ETA: 0s - loss: 0.2344 - mse: 2.350 - 0s 170us/sample - loss: 0.2260 - mse: 2.3335 - val_loss: 0.2142 - val_mse: 11.0840
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2982 - mse: 3.190 - ETA: 0s - loss: 0.1869 - mse: 2.315 - 0s 159us/sample - loss: 0.1853 - mse: 2.2941 - val_loss: 0.2142 - val_mse: 14.1641
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1275 - mse: 2.577 - ETA: 0s - loss: 0.1654 - mse: 2.285 - 0s 150us/sample - loss: 0.1606 - mse: 2.3215 - val_loss: 0.2142 - val_mse: 16.5521
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.2502 - mse: 2.687 - ETA: 0s - loss: 0.1418 - mse: 2.217 - 0s 134us/sample - loss: 0.1458 - mse: 2.3336 - val_loss: 0.2142 - val_mse: 18.3576
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.2216 - mse: 2.749 - ETA: 0s - loss: 0.1338 - mse: 2.342 - 0s 140us/sample - loss: 0.1364 - mse: 2.3676 - val_loss: 0.2142 - val_mse: 19.6279
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.0993 - mse: 1.318 - 0s 103us/sample - loss: 0.1302 - mse: 2.3859 - val_loss: 0.2142 - val_mse: 20.4884
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1309 - mse: 3.163 - ETA: 0s - loss: 0.1287 - mse: 2.395 - 0s 125us/sample - loss: 0.1258 - mse: 2.3845 - val_loss: 0.2142 - val_mse: 21.1285
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.0708 - mse: 1.045 - 0s 141us/sample - loss: 0.1225 - mse: 2.3942 - val_loss: 0.2142 - val_mse: 21.5549
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 11s - loss: 0.1152 - mse: 1.56 - 1s 1ms/sample - loss: 0.1340 - mse: 1.4337 - val_loss: 0.2387 - val_mse: 15.1861
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2146 - mse: 2.789 - 0s 105us/sample - loss: 0.1232 - mse: 1.4449 - val_loss: 0.2343 - val_mse: 16.7029
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1791 - mse: 1.293 - 0s 97us/sample - loss: 0.1168 - mse: 1.4632 - val_loss: 0.2313 - val_mse: 17.6525
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.0625 - mse: 1.063 - 0s 98us/sample - loss: 0.1125 - mse: 1.4683 - val_loss: 0.2293 - val_mse: 18.5404
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1017 - mse: 1.157 - 0s 104us/sample - loss: 0.1095 - mse: 1.4635 - val_loss: 0.2278 - val_mse: 19.1628
```

```
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.1535 - mse: 1.116 - 0s 114us/sample - loss: 0.1073 - mse:
1.4673 - val_loss: 0.2267 - val_mse: 19.6844
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.1605 - mse: 1.796 - 0s 105us/sample - loss: 0.1054 - mse:
1.4721 - val_loss: 0.2257 - val_mse: 19.9802
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.1295 - mse: 2.297 - 0s 118us/sample - loss: 0.1039 - mse:
1.4636 - val_loss: 0.2248 - val_mse: 20.2435
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1547 - mse: 3.121 - 0s 109us/sample - loss: 0.1026 - mse:
1.4596 - val_loss: 0.2241 - val_mse: 20.3862
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.0720 - mse: 1.528 - ETA: 0s - loss: 0.0999 - mse: 1.446 -
0s 136us/sample - loss: 0.1015 - mse: 1.4560 - val_loss: 0.2235 - val_mse: 20.4760
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 15s - loss: 0.1417 - mse: 2.22 - 1s 1ms/sample - loss: 0.1631 - mse:
2.6476 - val_loss: 0.2142 - val_mse: 17.4176
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.3243 - mse: 4.395 - 0s 105us/sample - loss: 0.1613 - mse:
2.6647 - val_loss: 0.2142 - val_mse: 17.7705
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1627 - mse: 2.931 - 0s 90us/sample - loss: 0.1600 - mse:
2.6685 - val_loss: 0.2142 - val_mse: 18.0416
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2778 - mse: 3.680 - 0s 93us/sample - loss: 0.1589 - mse:
2.6669 - val_loss: 0.2142 - val_mse: 18.2795
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.0927 - mse: 2.204 - 0s 94us/sample - loss: 0.1580 - mse:
2.6695 - val_loss: 0.2142 - val_mse: 18.4466
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.3625 - mse: 5.350 - 0s 107us/sample - loss: 0.1571 - mse:
2.6534 - val_loss: 0.2142 - val_mse: 18.6982
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.0545 - mse: 1.385 - 0s 100us/sample - loss: 0.1562 - mse:
2.6450 - val_loss: 0.2142 - val_mse: 18.8721
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.0135 - mse: 1.371 - 0s 94us/sample - loss: 0.1553 - mse:
2.6324 - val_loss: 0.2142 - val_mse: 19.0736
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.2039 - mse: 2.646 - 0s 90us/sample - loss: 0.1543 - mse:
2.6133 - val_loss: 0.2142 - val_mse: 19.2449
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.2501 - mse: 3.994 - 0s 93us/sample - loss: 0.1535 - mse:
2.5933 - val_loss: 0.2142 - val_mse: 19.3571
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.01

|-units: 12

|-Score: 0.21729836497041913

|-Best step: 0

```
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 10s - loss: 0.2651 - mse: 2.17 - 1s 1ms/sample - loss: 0.3101 - mse:
2.6623 - val_loss: 2.4770 - val_mse: 24.6645
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.3102 - mse: 2.894 - 0s 140us/sample - loss: 0.3027 - mse:
2.6381 - val_loss: 2.3992 - val_mse: 23.1556
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.2140 - mse: 1.424 - 0s 150us/sample - loss: 0.2961 - mse:
2.6197 - val_loss: 2.3233 - val_mse: 21.7476
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2880 - mse: 2.605 - ETA: 0s - loss: 0.2770 - mse: 2.468 -
0s 207us/sample - loss: 0.2898 - mse: 2.6023 - val_loss: 2.2501 - val_mse: 20.4467
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.2619 - mse: 2.675 - ETA: 0s - loss: 0.2812 - mse: 2.608 -
0s 201us/sample - loss: 0.2831 - mse: 2.5887 - val_loss: 2.1737 - val_mse: 19.1451
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.2500 - mse: 2.177 - ETA: 0s - loss: 0.2488 - mse: 2.231 -
0s 241us/sample - loss: 0.2768 - mse: 2.5788 - val_loss: 2.1012 - val_mse: 17.9638
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.3181 - mse: 2.400 - ETA: 0s - loss: 0.2690 - mse: 2.499 -
0s 198us/sample - loss: 0.2711 - mse: 2.5705 - val_loss: 2.0307 - val_mse: 16.8641
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.4406 - mse: 3.150 - 0s 161us/sample - loss: 0.2657 - mse:
2.5655 - val_loss: 1.9604 - val_mse: 15.8139
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.3042 - mse: 3.496 - 0s 143us/sample - loss: 0.2606 - mse:
2.5598 - val_loss: 1.8931 - val_mse: 14.8520
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.2542 - mse: 1.847 - 0s 145us/sample - loss: 0.2559 - mse:
2.5572 - val_loss: 1.8284 - val_mse: 13.9639
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 9s - loss: 0.2335 - mse: 1.488 - ETA: 0s - loss: 0.3022 - mse: 2.306 -
1s 1ms/sample - loss: 0.3024 - mse: 2.3003 - val_loss: 0.8833 - val_mse: 4.6209
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2335 - mse: 1.491 - ETA: 0s - loss: 0.2867 - mse: 2.195 -
0s 184us/sample - loss: 0.2938 - mse: 2.2656 - val_loss: 0.8117 - val_mse: 4.1825
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.3157 - mse: 2.691 - 0s 152us/sample - loss: 0.2858 - mse:
2.2319 - val_loss: 0.7439 - val_mse: 3.7937
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2200 - mse: 1.295 - 0s 146us/sample - loss: 0.2784 - mse:
2.2038 - val_loss: 0.6813 - val_mse: 3.4569
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.2627 - mse: 2.647 - ETA: 0s - loss: 0.2775 - mse: 2.209 -
0s 191us/sample - loss: 0.2717 - mse: 2.1754 - val_loss: 0.6233 - val_mse: 3.1624
```

```
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.2704 - mse: 2.276 - ETA: 0s - loss: 0.2544 - mse: 2.095 -
0s 195us/sample - loss: 0.2654 - mse: 2.1505 - val_loss: 0.5678 - val_mse: 2.8963
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.2269 - mse: 1.175 - 0s 141us/sample - loss: 0.2595 - mse:
2.1287 - val_loss: 0.5169 - val_mse: 2.6648
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.1824 - mse: 1.251 - 0s 145us/sample - loss: 0.2540 - mse:
2.1104 - val_loss: 0.4699 - val_mse: 2.4613
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1754 - mse: 1.726 - ETA: 0s - loss: 0.2444 - mse: 2.034 -
0s 172us/sample - loss: 0.2489 - mse: 2.0906 - val_loss: 0.4271 - val_mse: 2.2836
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.2354 - mse: 2.224 - 0s 141us/sample - loss: 0.2442 - mse:
2.0748 - val_loss: 0.3890 - val_mse: 2.1304
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 9s - loss: 0.3172 - mse: 1.763 - 1s 1ms/sample - loss: 0.2506 - mse:
1.5827 - val_loss: 0.2533 - val_mse: 6.0947
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2711 - mse: 1.673 - 0s 154us/sample - loss: 0.2371 - mse:
1.5390 - val_loss: 0.2421 - val_mse: 6.5102
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1921 - mse: 1.195 - 0s 187us/sample - loss: 0.2248 - mse:
1.5018 - val_loss: 0.2326 - val_mse: 6.9277
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2065 - mse: 1.367 - 0s 154us/sample - loss: 0.2139 - mse:
1.4674 - val_loss: 0.2246 - val_mse: 7.3356
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1593 - mse: 0.970 - 0s 147us/sample - loss: 0.2044 - mse:
1.4408 - val_loss: 0.2175 - val_mse: 7.7353
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.2328 - mse: 1.282 - ETA: 0s - loss: 0.1924 - mse: 1.365 -
0s 187us/sample - loss: 0.1959 - mse: 1.4196 - val_loss: 0.2115 - val_mse: 8.1215
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.1490 - mse: 0.973 - 0s 154us/sample - loss: 0.1883 - mse:
1.4008 - val_loss: 0.2061 - val_mse: 8.5106
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.1552 - mse: 0.876 - 0s 149us/sample - loss: 0.1815 - mse:
1.3857 - val_loss: 0.2014 - val_mse: 8.8968
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1705 - mse: 1.201 - ETA: 0s - loss: 0.1735 - mse: 1.371 -
0s 219us/sample - loss: 0.1753 - mse: 1.3735 - val_loss: 0.1974 - val_mse: 9.2647
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.2586 - mse: 1.666 - ETA: 0s - loss: 0.1719 - mse: 1.296 -
0s 189us/sample - loss: 0.1699 - mse: 1.3649 - val_loss: 0.1940 - val_mse: 9.6145
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.001

|-units: 20

|-Score: 0.8037920699516933

|-Best step: 0

```
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 10s - loss: 0.1684 - mse: 1.06 - ETA: 0s - loss: 0.2880 - mse: 1.8078
- 1s 1ms/sample - loss: 0.2879 - mse: 1.8118 - val_loss: 0.2142 - val_mse: 5.8789
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2582 - mse: 1.399 - ETA: 0s - loss: 0.2858 - mse: 1.818 -
0s 166us/sample - loss: 0.2772 - mse: 1.7584 - val_loss: 0.2142 - val_mse: 6.6973
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.2301 - mse: 1.589 - ETA: 0s - loss: 0.2728 - mse: 1.727 -
0s 158us/sample - loss: 0.2672 - mse: 1.7110 - val_loss: 0.2142 - val_mse: 7.5712
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.1561 - mse: 0.957 - ETA: 0s - loss: 0.2541 - mse: 1.594 -
0s 160us/sample - loss: 0.2579 - mse: 1.6655 - val_loss: 0.2142 - val_mse: 8.5069
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.2213 - mse: 1.196 - 0s 107us/sample - loss: 0.2491 - mse:
1.6255 - val_loss: 0.2142 - val_mse: 9.4815
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.3181 - mse: 2.045 - 0s 104us/sample - loss: 0.2409 - mse:
1.5899 - val_loss: 0.2142 - val_mse: 10.4935
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.2379 - mse: 1.709 - 0s 125us/sample - loss: 0.2332 - mse:
1.5573 - val_loss: 0.2142 - val_mse: 11.5392
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.2155 - mse: 1.319 - 0s 164us/sample - loss: 0.2259 - mse:
1.5280 - val_loss: 0.2142 - val_mse: 12.6076
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.2162 - mse: 1.915 - ETA: 0s - loss: 0.2185 - mse: 1.522 -
0s 144us/sample - loss: 0.2192 - mse: 1.5015 - val_loss: 0.2142 - val_mse: 13.7033
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.2722 - mse: 1.578 - ETA: 0s - loss: 0.2209 - mse: 1.533 -
0s 153us/sample - loss: 0.2128 - mse: 1.4781 - val_loss: 0.2142 - val_mse: 14.8050
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 12s - loss: 0.5751 - mse: 4.01 - 1s 1ms/sample - loss: 0.5541 - mse:
3.5412 - val_loss: 4.1518 - val_mse: 94.2987
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.1860 - mse: 0.901 - 0s 104us/sample - loss: 0.5354 - mse:
3.4029 - val_loss: 4.0642 - val_mse: 89.6174
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.2453 - mse: 1.089 - 0s 123us/sample - loss: 0.5175 - mse:
3.2755 - val_loss: 3.9774 - val_mse: 85.1492
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.6303 - mse: 4.502 - 0s 103us/sample - loss: 0.5005 - mse:
3.1563 - val_loss: 3.8916 - val_mse: 80.8862
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.6183 - mse: 4.296 - 0s 110us/sample - loss: 0.4840 - mse:
3.0440 - val_loss: 3.8059 - val_mse: 76.7925
```

```
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.5311 - mse: 2.882 - ETA: 0s - loss: 0.4763 - mse: 3.000 -
0s 133us/sample - loss: 0.4682 - mse: 2.9377 - val_loss: 3.7216 - val_mse: 72.9088
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.5509 - mse: 3.114 - 0s 116us/sample - loss: 0.4530 - mse:
2.8379 - val_loss: 3.6380 - val_mse: 69.2117
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.4358 - mse: 2.661 - 0s 103us/sample - loss: 0.4384 - mse:
2.7452 - val_loss: 3.5556 - val_mse: 65.7013
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.3149 - mse: 1.724 - 0s 121us/sample - loss: 0.4245 - mse:
2.6576 - val_loss: 3.4744 - val_mse: 62.3672
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.4545 - mse: 2.781 - ETA: 0s - loss: 0.4116 - mse: 2.574 -
0s 140us/sample - loss: 0.4111 - mse: 2.5775 - val_loss: 3.3946 - val_mse: 59.2071
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 15s - loss: 0.3293 - mse: 1.93 - ETA: 0s - loss: 0.2918 - mse: 1.6378
- 1s 1ms/sample - loss: 0.2930 - mse: 1.6461 - val_loss: 1.7003 - val_mse: 14.0277
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2150 - mse: 0.885 - ETA: 0s - loss: 0.2724 - mse: 1.552 -
0s 156us/sample - loss: 0.2741 - mse: 1.5723 - val_loss: 1.6080 - val_mse: 12.7749
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.2257 - mse: 1.257 - ETA: 0s - loss: 0.2387 - mse: 1.373 -
0s 152us/sample - loss: 0.2570 - mse: 1.5053 - val_loss: 1.5194 - val_mse: 11.6409
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.2415 - mse: 1.205 - ETA: 0s - loss: 0.2423 - mse: 1.469 -
0s 144us/sample - loss: 0.2419 - mse: 1.4473 - val_loss: 1.4355 - val_mse: 10.6255
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1693 - mse: 0.830 - 0s 125us/sample - loss: 0.2285 - mse:
1.3974 - val_loss: 1.3558 - val_mse: 9.7135
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.2309 - mse: 1.129 - ETA: 0s - loss: 0.2160 - mse: 1.344 -
0s 145us/sample - loss: 0.2167 - mse: 1.3524 - val_loss: 1.2805 - val_mse: 8.8974
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.1792 - mse: 1.061 - ETA: 0s - loss: 0.2046 - mse: 1.287 -
0s 152us/sample - loss: 0.2064 - mse: 1.3129 - val_loss: 1.2102 - val_mse: 8.1732
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.1709 - mse: 1.278 - 0s 160us/sample - loss: 0.1976 - mse:
1.2794 - val_loss: 1.1447 - val_mse: 7.5321
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1772 - mse: 1.384 - 0s 125us/sample - loss: 0.1899 - mse:
1.2504 - val_loss: 1.0833 - val_mse: 6.9587
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.1624 - mse: 0.784 - 0s 119us/sample - loss: 0.1833 - mse:
1.2242 - val_loss: 1.0266 - val_mse: 6.4530
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.001

|-units: 14

|-Score: 1.545140876173973

|-Best step: 0

```
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 9s - loss: 0.5462 - mse: 3.590 - 1s 913us/sample - loss: 0.3963 - mse:
2.7392 - val_loss: 1.2289 - val_mse: 7.7978
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.3283 - mse: 2.158 - ETA: 0s - loss: 0.3920 - mse: 2.714 -
0s 185us/sample - loss: 0.3948 - mse: 2.7304 - val_loss: 1.2232 - val_mse: 7.7457
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.2493 - mse: 1.629 - 0s 144us/sample - loss: 0.3933 - mse:
2.7214 - val_loss: 1.2175 - val_mse: 7.6942
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.3017 - mse: 1.837 - 0s 139us/sample - loss: 0.3918 - mse:
2.7125 - val_loss: 1.2119 - val_mse: 7.6433
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1810 - mse: 1.235 - 0s 148us/sample - loss: 0.3903 - mse:
2.7038 - val_loss: 1.2063 - val_mse: 7.5928
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.3773 - mse: 2.363 - ETA: 0s - loss: 0.3683 - mse: 2.512 -
0s 188us/sample - loss: 0.3888 - mse: 2.6953 - val_loss: 1.2006 - val_mse: 7.5426
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.5643 - mse: 4.109 - 0s 160us/sample - loss: 0.3874 - mse:
2.6869 - val_loss: 1.1951 - val_mse: 7.4932
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.2708 - mse: 1.613 - 0s 157us/sample - loss: 0.3859 - mse:
2.6784 - val_loss: 1.1896 - val_mse: 7.4443
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.3981 - mse: 2.541 - ETA: 0s - loss: 0.3962 - mse: 2.725 -
0s 186us/sample - loss: 0.3845 - mse: 2.6700 - val_loss: 1.1841 - val_mse: 7.3958
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.6233 - mse: 4.472 - 0s 135us/sample - loss: 0.3830 - mse:
2.6618 - val_loss: 1.1786 - val_mse: 7.3478
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 10s - loss: 0.1572 - mse: 2.94 - ETA: 0s - loss: 0.1636 - mse: 3.8171
- 1s 1ms/sample - loss: 0.1583 - mse: 3.6636 - val_loss: 0.2071 - val_mse: 2.2801
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.3037 - mse: 5.404 - 0s 114us/sample - loss: 0.1583 - mse:
3.6639 - val_loss: 0.2071 - val_mse: 2.2826
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1428 - mse: 3.473 - ETA: 0s - loss: 0.1621 - mse: 3.836 -
0s 187us/sample - loss: 0.1582 - mse: 3.6641 - val_loss: 0.2071 - val_mse: 2.2850
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.0842 - mse: 2.139 - ETA: 0s - loss: 0.1621 - mse: 3.745 -
0s 128us/sample - loss: 0.1582 - mse: 3.6644 - val_loss: 0.2072 - val_mse: 2.2875
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.1412 - mse: 3.576 - 0s 111us/sample - loss: 0.1582 - mse:
3.6647 - val_loss: 0.2072 - val_mse: 2.2899
```

```
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.0535 - mse: 1.353 - 0s 115us/sample - loss: 0.1581 - mse: 3.6650 - val_loss: 0.2072 - val_mse: 2.2923
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.0713 - mse: 2.274 - ETA: 0s - loss: 0.1558 - mse: 3.729 - 0s 128us/sample - loss: 0.1581 - mse: 3.6653 - val_loss: 0.2072 - val_mse: 2.2948
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.0890 - mse: 3.854 - 0s 118us/sample - loss: 0.1581 - mse: 3.6654 - val_loss: 0.2072 - val_mse: 2.2972
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.2913 - mse: 4.875 - 0s 115us/sample - loss: 0.1580 - mse: 3.6657 - val_loss: 0.2073 - val_mse: 2.2996
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.1627 - mse: 4.327 - 0s 120us/sample - loss: 0.1580 - mse: 3.6660 - val_loss: 0.2073 - val_mse: 2.3020
Train on 700 samples, validate on 300 samples
Epoch 1/10
700/700 [==============================] - ETA: 10s - loss: 0.0654 - mse: 4.54 - 1s 994us/sample - loss: 0.1929 - mse: 4.9452 - val_loss: 0.2142 - val_mse: 7.3170
Epoch 2/10
700/700 [==============================] - ETA: 0s - loss: 0.2105 - mse: 5.969 - 0s 114us/sample - loss: 0.1928 - mse: 4.9463 - val_loss: 0.2142 - val_mse: 7.3343
Epoch 3/10
700/700 [==============================] - ETA: 0s - loss: 0.1286 - mse: 2.833 - 0s 114us/sample - loss: 0.1927 - mse: 4.9477 - val_loss: 0.2142 - val_mse: 7.3515
Epoch 4/10
700/700 [==============================] - ETA: 0s - loss: 0.1573 - mse: 5.448 - 0s 104us/sample - loss: 0.1926 - mse: 4.9488 - val_loss: 0.2142 - val_mse: 7.3688
Epoch 5/10
700/700 [==============================] - ETA: 0s - loss: 0.2316 - mse: 3.835 - 0s 108us/sample - loss: 0.1925 - mse: 4.9502 - val_loss: 0.2142 - val_mse: 7.3861
Epoch 6/10
700/700 [==============================] - ETA: 0s - loss: 0.1341 - mse: 3.091 - ETA: 0s - loss: 0.1975 - mse: 5.041 - 0s 127us/sample - loss: 0.1924 - mse: 4.9515 - val_loss: 0.2142 - val_mse: 7.4033
Epoch 7/10
700/700 [==============================] - ETA: 0s - loss: 0.2731 - mse: 4.268 - ETA: 0s - loss: 0.1869 - mse: 4.679 - 0s 151us/sample - loss: 0.1923 - mse: 4.9528 - val_loss: 0.2142 - val_mse: 7.4205
Epoch 8/10
700/700 [==============================] - ETA: 0s - loss: 0.1058 - mse: 4.611 - ETA: 0s - loss: 0.2017 - mse: 4.885 - 0s 134us/sample - loss: 0.1922 - mse: 4.9541 - val_loss: 0.2142 - val_mse: 7.4376
Epoch 9/10
700/700 [==============================] - ETA: 0s - loss: 0.1310 - mse: 4.517 - 0s 108us/sample - loss: 0.1922 - mse: 4.9551 - val_loss: 0.2142 - val_mse: 7.4548
Epoch 10/10
700/700 [==============================] - ETA: 0s - loss: 0.1541 - mse: 4.405 - ETA: 0s - loss: 0.1910 - mse: 4.962 - 0s 150us/sample - loss: 0.1921 - mse: 4.9565 - val_loss: 0.2142 - val_mse: 7.4718
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.0001

|-units: 16

|-Score: 0.5333168996042675

|-Best step: 0

In [48]:
```
## Print out the summary results of our data mining process using Keras

tuner.results_summary()
```

Results summary

|-Results in C:\untitled_project

|-Showing 10 best trials

|-Objective: Objective(name='val_loss', direction='min') Score: 0.21729836497041913

|-Objective: Objective(name='val_loss', direction='min') Score: 0.22868611564238864

|-Objective: Objective(name='val_loss', direction='min') Score: 0.5333168996042675

|-Objective: Objective(name='val_loss', direction='min') Score: 0.8037920699516933

|-Objective: Objective(name='val_loss', direction='min') Score: 1.545140876173973

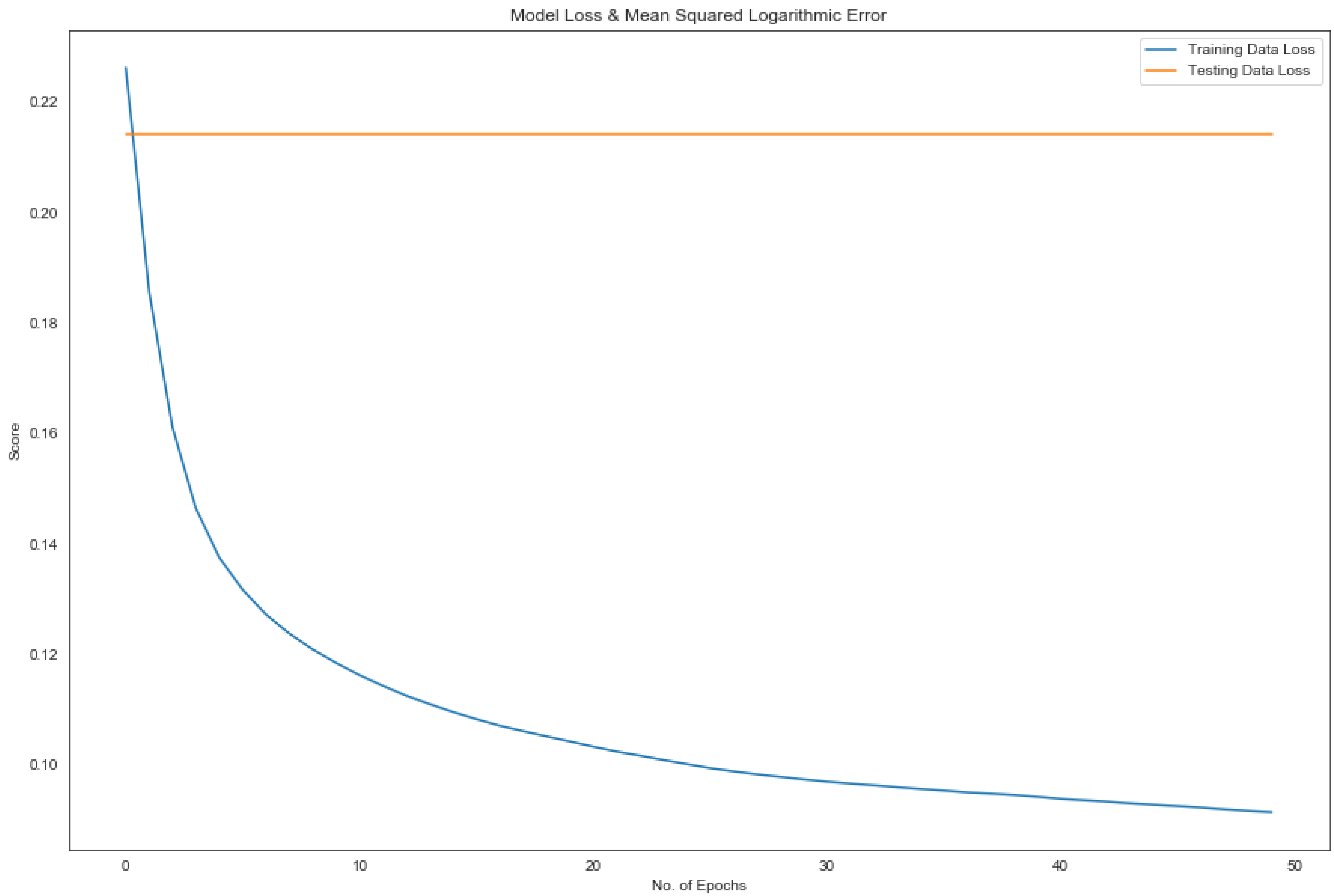
In [50]:
```
## Save the best models to evaluate their performance below

models = tuner.get_best_models(num_models = 2)
```

In [51]:
```
## Save the single best model to a new variable

best_model = models[0]
```

In [53]:
```
### Use our best model and train it again

non_spend_history = best_model.fit(X_train_standard, y_train_standard,
                validation_data = (X_test_standard, y_test_standard), epochs = 50, verbose = 0)
```

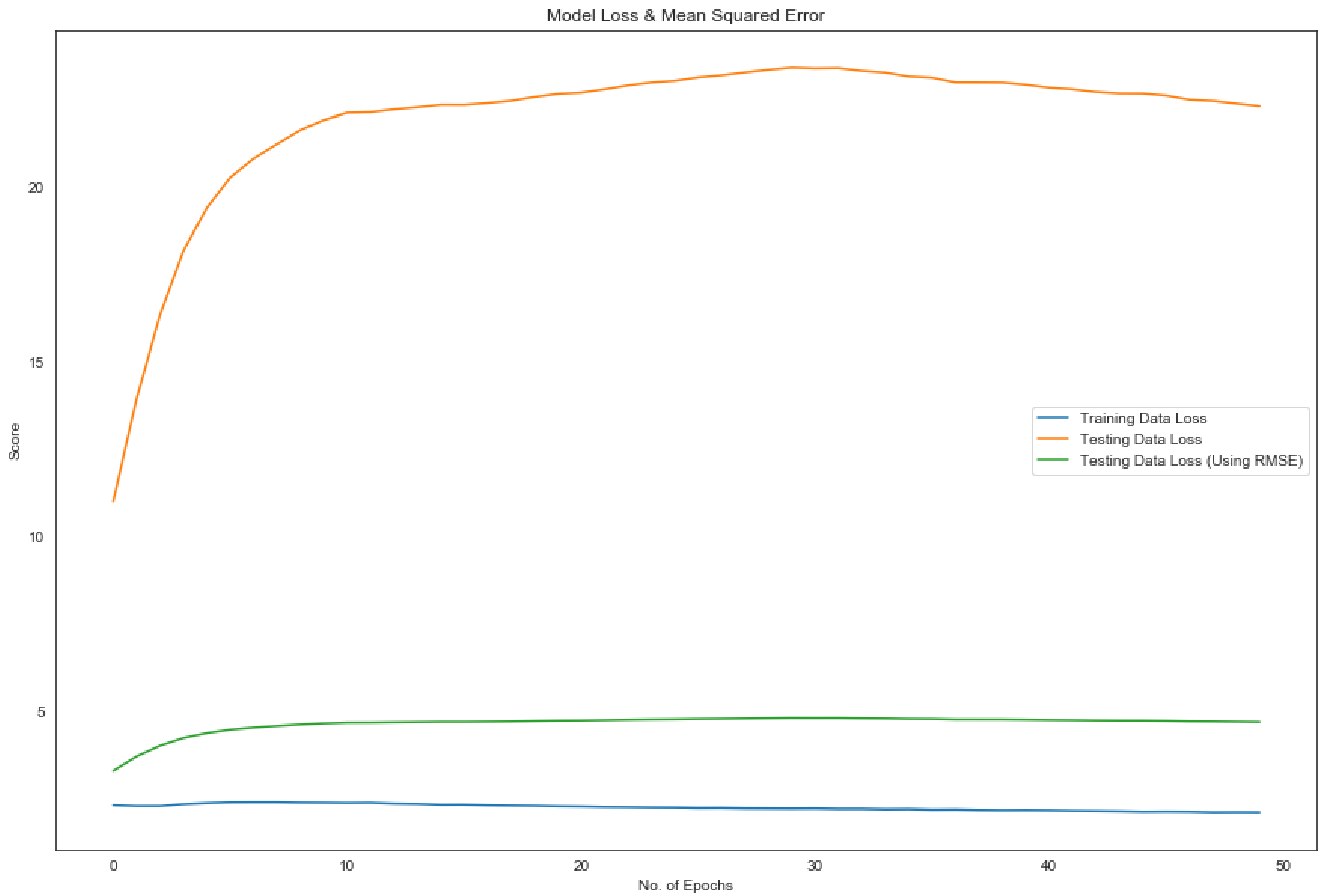
In [54]:
```
## Evaluate the "best" model

train_mse = model.evaluate(X_train_standard, y_train_standard, verbose=0)
test_mse = model.evaluate(X_test_standard, y_test_standard, verbose=0)
```

In [56]:
```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot the MSLE for this model

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot the training data "loss" compared to the testing data "loss"
plt.plot(non_spend_history.history['loss'], label = "Training Data Loss")
plt.plot(non_spend_history.history['val_loss'], label = "Testing Data Loss")

## Provide some labels
plt.xlabel("No. of Epochs")
plt.ylabel("Score")

## Provide a title and legend
plt.title("Model Loss & Mean Squared Logarithmic Error")
plt.legend()

## Show the graph
plt.show()
```


Model Loss & Mean Squared Logarithmic Error
Training Data Loss
Testing Data Loss
Score
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0
10
20
30
40
50
No. of Epochs

In [58]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot the training data "loss" compared to the testing data "loss"
plt.plot(non_spend_history.history['mse'], label = "Training Data Loss")
plt.plot(non_spend_history.history['val_mse'], label = "Testing Data Loss")

## Add more line for plotting the RMSE of testing data
plt.plot([sqrt(i) for i in non_spend_history.history['val_mse']], label = "Testing Data Loss (Using RMSE)")

## Provide some labels
plt.xlabel("No. of Epochs")
plt.ylabel("Score")

## Provide a title and legend
plt.title("Model Loss & Mean Squared Error")
plt.legend(loc = "right")

## Show the graph
plt.show()
```



D. Conclusions / Model Evaluation

From the various models tested, it seems the best predictor of "Spending" is the model that was developed using Keras & Keras Tuner. The reasoning behind this is that the model has less training data to use to learn, since we are removing a lot of the variance in the data by subsetting it to only include Purchases.

In our original explorations, we were using training data that had a lot of 0.00 values in the Spending column (no purchase). This meant that our modeling techniques were struggling with identifying the actual purchasing behavior because there were a lot of false positives. The mining technique would see two sets of features that were similar but one would end with spend and one would not, and do a poor job assigning or estimating the spend of a customer.

The basic idea is that there are two possible (and almost opposite) reasons for a data mining technique to not perform well.

In the first case, we might have a model that is too complicated for the amount of data we have. This situation, known as high variance, leads to model over-fitting. We know that we are facing a high variance problem with this data set, and it is confirmed when we see that the training error is much lower than testing error.

High variance problems can be addressed by reducing the number of features, and... yes, by increasing the number of data points, which we can't do here.

These models have many features, as compared to our training examples, which is why we see the over-fitting in the first attempt of using Keras and Keras Tuner.

However, we might have a model that is too simple to explain the data we have. In this case, which is known as high bias, adding more data will not help.

Here is the specific model selection and model testing steps. The "best_model" would be declared the winner!

```
## Run our search with Keras Tuner!

tuner.search(X_train_standard, y_train_standard, epochs = 10,
             validation_data = (X_test_standard, y_test_standard))
```

...

```
## Print out the summary results of our data mining process using Keras

tuner.results_summary()
```

...

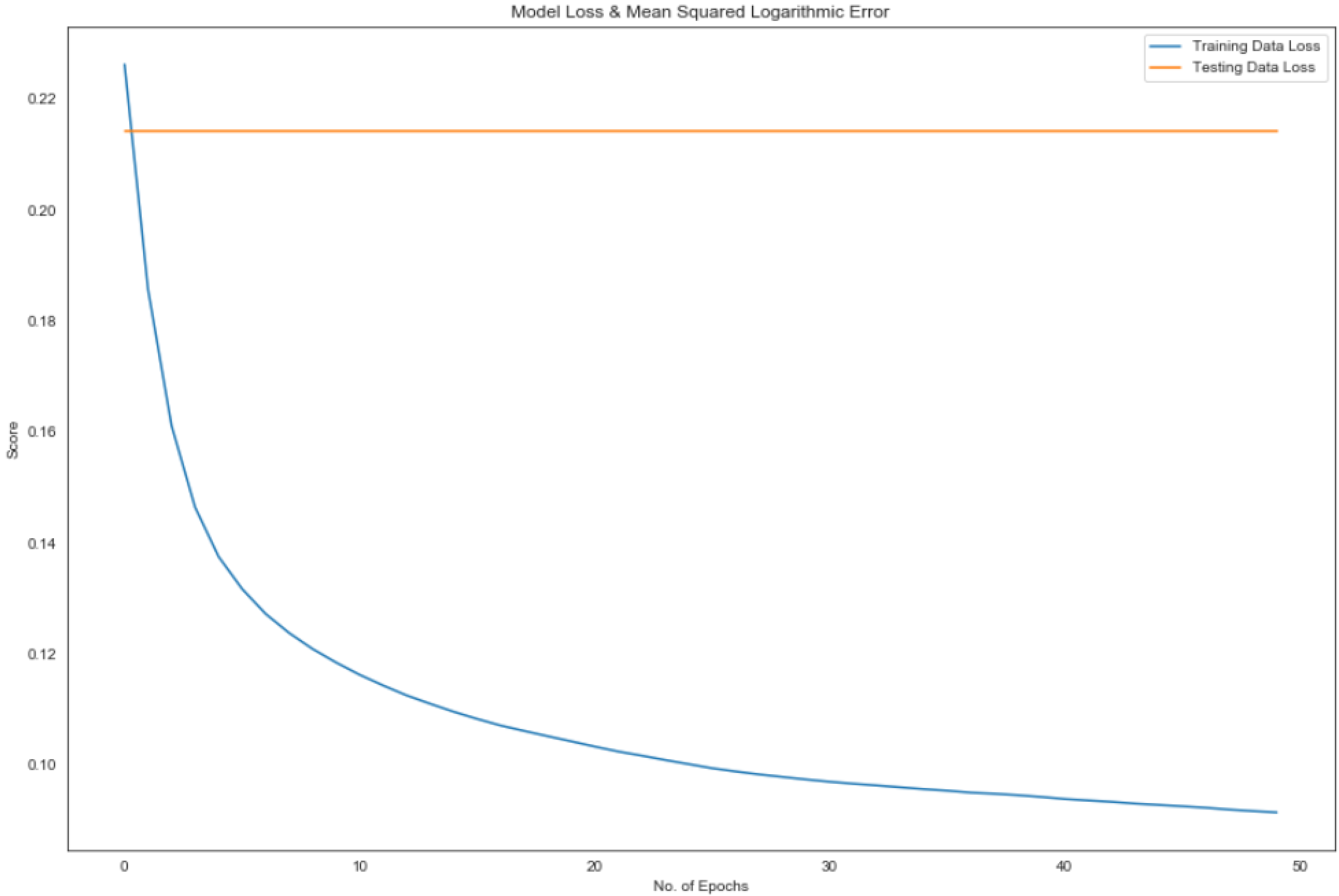
```
## Save the best models to evaluate their performance below

models = tuner.get_best_models(num_models = 2)
```

```
## Save the single best model to a new variable

best_model = models[0]
```

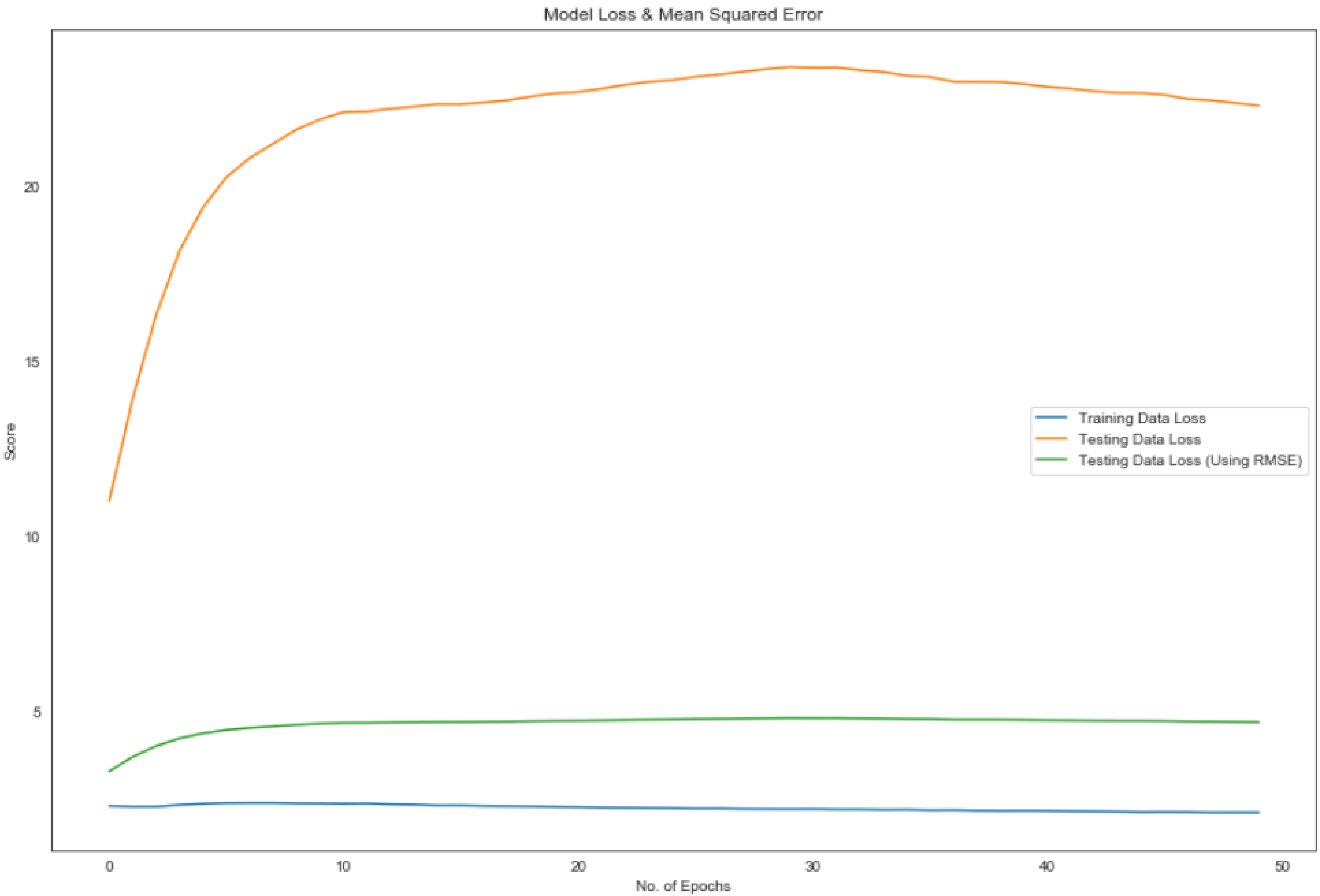
```
### Use our best model and train it again

non_spend_history = best_model.fit(X_train_standard, y_train_standard,
                validation_data = (X_test_standard, y_test_standard), epochs = 50, verbose = 0)
```

```
## Evaluate the "best" model

train_mse = model.evaluate(X_train_standard, y_train_standard, verbose=0)
test_mse = model.evaluate(X_test_standard, y_test_standard, verbose=0)
```

I am highlighting two of the plots that were generated, that show a nice smoothing of the training data loss, which meant that the model was learning at a steady rate. This is unlike the first attempt of building a model with Keras, where we saw a lot of uneven learning.


Model Loss & Mean Squared Logarithmic Error
Training Data Loss
Testing Data Loss
Score
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0
10
20
30
40
50
No. of Epochs

When I look at the RMSE for the preferred regression model, it is much more stable and doesn't fluctuate quite as much when going through training cycles. I am confident that this model is performing much better compared to my earlier attempts with the "out-of-box" regressors from `sklearn`.



Model Loss & Mean Squared Error
Score
No. of Epochs
Training Data Loss
Testing Data Loss
Testing Data Loss (Using RMSE)
0
10
20
30
40
50
5
10
15
20

Quation 2. (50 points) Download the dataset on spam vs. non-spam emails from the following URL: http://archive.ics.uci.edu/ml/datasets/Spambase. Specifically, (i) file "spambase.data" contains the actual data, and (ii) files "spambase.names" and "spambase.DOCUMENTATION" contain the description of the data. This dataset has 4601 records, each record representing a different email message. Each record is described with 58 attributes (indicated in the aforementioned .names file): attributes 1-57 represent various content-based characteristics already extracted from each email message (related to the frequency of certain words or certain punctuation symbols in a message as well as to the usage of capital letters in a message), and the last attribute represents the class label for each message (spam or non-spam).

```
SPAM E-MAIL DATABASE ATTRIBUTES (in .names format)

48 continuous real [0,100] attributes of type word_freq_WORD
= percentage of words in the e-mail that match WORD,
i.e. 100 * (number of times the WORD appears in the e-mail) /
total number of words in e-mail.  A "word" in this case is any
string of alphanumeric characters bounded by non-alphanumeric
characters or end-of-string.

6 continuous real [0,100] attributes of type char_freq_CHAR
= percentage of characters in the e-mail that match CHAR,
i.e. 100 * (number of CHAR occurences) / total characters in e-mail

1 continuous real [1,...] attribute of type capital_run_length_average
= average length of uninterrupted sequences of capital letters

1 continuous integer [1,...] attribute of type capital_run_length_longest
= length of longest uninterrupted sequence of capital letters

1 continuous integer [1,...] attribute of type capital_run_length_total
= sum of length of uninterrupted sequences of capital letters
= total number of capital letters in the e-mail

1 nominal {0,1} class attribute of type spam
= denotes whether the e-mail was considered spam (1) or not (0),
i.e. unsolicited commercial e-mail.

For more information, see file 'spambase.DOCUMENTATION' at the
UCI Machine Learning Repository: http://www.ics.uci.edu/~mlearn/MLRepository.html
```

A. Data pre-processing and pre-analysis

1. Read in the data
2. Explore the features and target variables to assess what parameters will need to be changed
3. Prepare & transform data for data mining process

In [3]:
```
## Pull in all the column names based on the spam base documentation

colnames = ["word_freq_make", "word_freq_address", "word_freq_all", "word_freq_3d", "word_freq_our",
            "word_freq_over", "word_freq_remove", "word_freq_internet", "word_freq_order", "word_freq_mail",
            "word_freq_receive", "word_freq_will", "word_freq_people", "word_freq_report", "word_freq_addresses",
            "word_freq_free", "word_freq_business", "word_freq_email", "word_freq_you", "word_freq_credit",
            "word_freq_your", "word_freq_font", "word_freq_000", "word_freq_money", "word_freq_hp",
            "word_freq_hpl", "word_freq_george", "word_freq_650", "word_freq_lab", "word_freq_labs",
            "word_freq_telnet", "word_freq_857", "word_freq_data", "word_freq_415", "word_freq_85",
            "word_freq_technology", "word_freq_1999", "word_freq_parts", "word_freq_pm", "word_freq_direct",
            "word_freq_cs", "word_freq_meeting", "word_freq_original", "word_freq_project", "word_freq_re",
            "word_freq_edu", "word_freq_table", "word_freq_conference", "char_freq_;", "char_freq_(",
            "char_freq_[", "char_freq_!", "char_freq_$", "char_freq_#", "capital_run_length_average",
            "capital_run_length_longest", "capital_run_length_total", "spam"]
```

In [4]:
```
## Read in the file and save it as a dataframe

spambase_df = pd.read_csv("spambase.data", header = None, names = colnames)
```

In [5]:
```
## Look at the df to make sure it loaded correctly

spambase_df.tail()
```

Out[5]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	wor
4596	0.31	0.0	0.62	0.0	0.00	0.31	0.0	0.0	
4597	0.00	0.0	0.00	0.0	0.00	0.00	0.0	0.0	
4598	0.30	0.0	0.30	0.0	0.00	0.00	0.0	0.0	
4599	0.96	0.0	0.00	0.0	0.32	0.00	0.0	0.0	
4600	0.00	0.0	0.65	0.0	0.00	0.00	0.0	0.0	

5 rows × 58 columns

In [6]:
```
## Generate some summary statistics of the dataframe, just to see the distribution of the various attributes

spambase_df.describe()
```

Out[6]:

	word_freq_make	word_freq_address	word_freq_all	word_freq_3d	word_freq_our	word_freq_over	word_freq_remove	word_freq_internet	wo
count	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	4601.000000	
mean	0.104553	0.213015	0.280656	0.065425	0.312223	0.095901	0.114208	0.105295	
std	0.305358	1.290575	0.504143	1.395151	0.672513	0.273824	0.391441	0.401071	
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
50%	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	
75%	0.000000	0.000000	0.420000	0.000000	0.380000	0.000000	0.000000	0.000000	
max	4.540000	14.280000	5.100000	42.810000	10.000000	5.880000	7.270000	11.110000	

8 rows × 58 columns

In [7]:
```
## Grab all columns as features except our target which is spam!

spam_feature_cols = spambase_df.columns[spambase_df.columns != "spam"]
```

In [8]:
```
spambase_df.spam.unique()
```

Out[8]:
```
array([1, 0], dtype=int64)
```

In [9]:
```
spambase_df.groupby("spam")["word_freq_make"].count()
```

Out[9]:
```
spam
0    2788
1    1813
Name: word_freq_make, dtype: int64
```

In [10]:
```
## View splits for the targets

total_obvs = spambase_df.groupby("spam")["word_freq_make"].count().sum()
total_non_spam = spambase_df.groupby("spam")["word_freq_make"].count()[0]
total_spam = spambase_df.groupby("spam")["word_freq_make"].count()[1]

## Uneven balanced classes
print("{} total non spam messages, \
     {} % of overall observations".format(total_non_spam, round(total_non_spam/total_obvs*100, 2)))
print("{} total spam messages, \
     {} % of overall observations".format(total_spam, round(total_spam/total_obvs*100, 2)))
```

```
2788 total non spam messages,      60.6 % of overall observations
1813 total spam messages,      39.4 % of overall observations
```

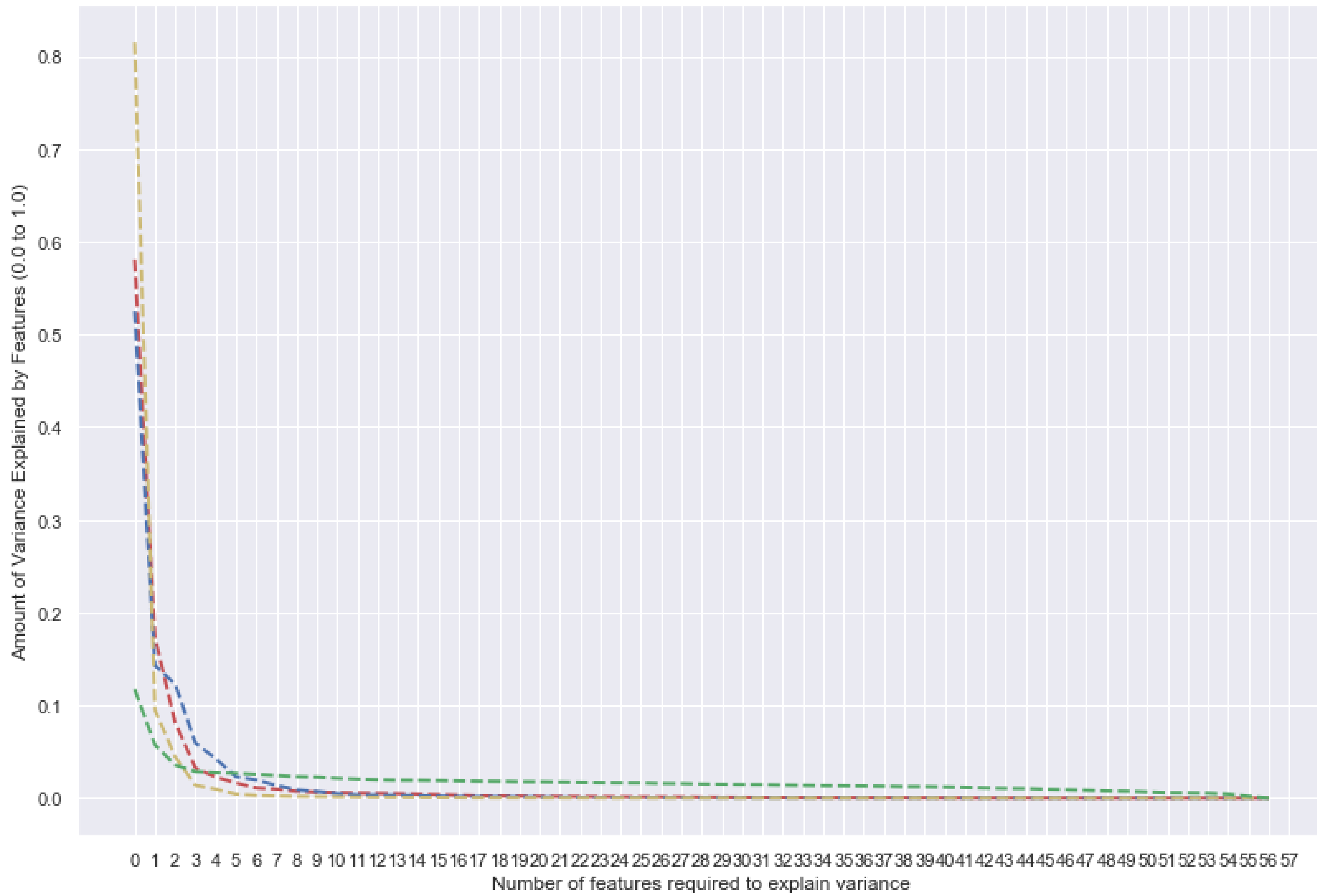
We have unbalanced classes, so we will `stratify` our splits before developing the model. For a classification task, this is chosen to ensure that the train and test sets have approximately the same percentage of samples of each target class as the complete data set.

In [12]:
```
## Grabbing all the features available
X = np.array(spambase_df[spam_feature_cols])

## Target variable
y = np.array(spambase_df["spam"])
```

In [13]:
```
## Split data training 70 % and testing 30%

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 42, stratify = y)
```

In [260]:
```
## We normalize our training and testing data PCA to work correctly
## We don't want to skew the results of the plot because some features are not on the same scale
## We perform this normalization AFTER splitting the data - again, so that we don't skew the training
## data with the testing data.

## Normalization is the process of scaling individual samples to have unit norm. This process can be useful
## if you plan to use a quadratic form such as the dot-product or any other kernel to quantify the similarity of any pa
ir of samples.
X_train_norm = Normalizer(norm = "l1").fit_transform(X_train)
X_train_robust = RobustScaler().fit_transform(X_train)
X_train_power = PowerTransformer(method='yeo-johnson', standardize=False).fit_transform(X_train)
X_train_standard = StandardScaler().fit_transform(X_train)

## Import PCA from sklearn
from sklearn.decomposition import PCA

## Initialize two new PCA instances, we'll use this to plot the training data using two different transformations
## Normalizer will likely be skewed by the outliers in each of the features being used
## Robust is going to transform feature values to be larger than the previous scalers and more importantly are approxim
ately similar to original data
## PowerTransformer is a family of parametric, monotonic transformations that aim to map data from any distribution to
 as close to a Gaussian distribution
## as possible in order to stabilize variance and minimize skewness.

pca = PCA()
pca_r = PCA()
pca_p = PCA()
pca_s = PCA()

## The goal of this plot is to determine what features need to be included in our models
## In the first few homeworks, we threw the kitchen sink at the models.  Here, we are going to be
## more refined in our analysis.

## Train the PCA instance using the normalized training data
pca.fit(X_train_norm)
pca_r.fit(X_train_robust)
pca_p.fit(X_train_power)
pca_s.fit(X_train_standard)

plt.figure(1, figsize=(15, 10))
plt.clf()
plt.axes([.2, .2, .7, .7])

plt.plot(pca.explained_variance_ratio_, 'r--', linewidth = 2)
plt.plot(pca_r.explained_variance_ratio_, 'b--', linewidth = 2)
plt.plot(pca_p.explained_variance_ratio_, 'y--', linewidth = 2)
plt.plot(pca_s.explained_variance_ratio_, 'g--', linewidth = 2)
```

```
## Set plot labels
plt.xlabel('Number of features required to explain variance')
plt.ylabel('Amount of Variance Explained by Features (0.0 to 1.0)')

## Explicitly set the x-axis data so we can see where the drop-off is
plt.xticks(np.arange(0, 58, step=1))

## Show the graph!
plt.show()
```

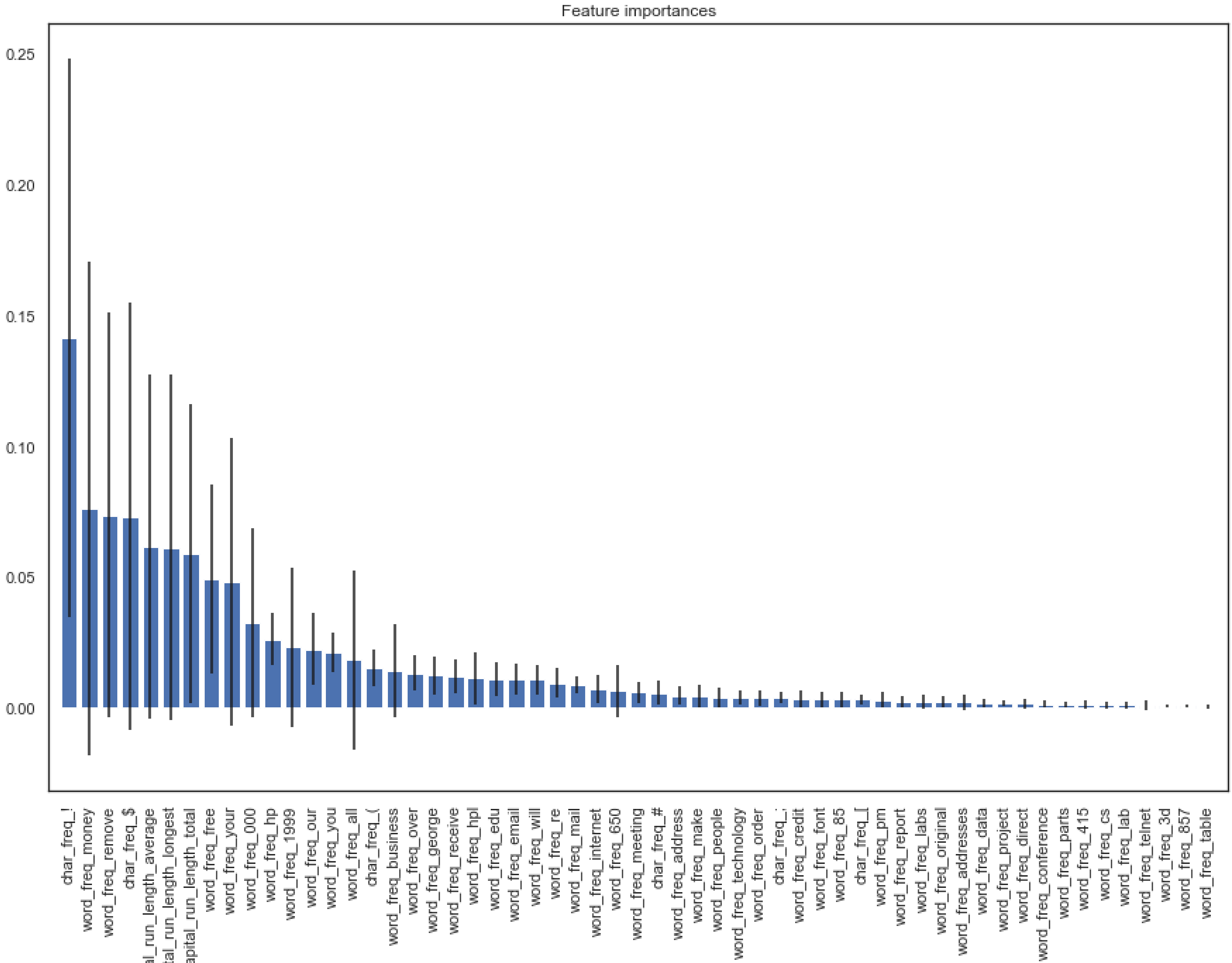


Only a very limited amount of features explain the variance in the data set. For some of the classifiers, more information might actually lead to worse performance, either through over-fitting or not being able to converge successfully.

In [267]:

```
## SET A VARIABLE TO SWAP IN AND OUT BETWEEN THE DIFFERENT TRAINING INSTANCES

#Z = X_train_robust
#Z = X_train_norm
#Z = X_train_power
Z = X_train_standard

## Start with identifying the best features using a Random Forest classifier

## Create a new classifier
clf_rf_5 = ensemble.RandomForestClassifier()
clr_rf_5 = clf_rf_5.fit(Z, y_train)

## Save our importances to a variable
importances = clr_rf_5.feature_importances_

## Get the standard deviation for each feature
std = np.std([tree.feature_importances_ for tree in clf_rf_5.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]

## Print the feature ranking
print("Feature ranking:")

## Print the top ten features, and their importance based on the Random Forest Classifier
for f in range(0, 10):
    print("%d. feature %d (%f)" % (f + 1, indices[f], importances[indices[f]]))


## Plot the feature importances of the Random Forest Regressor - to see this visually

## Set the plot size
plt.figure(1, figsize=(15, 10))

## Set the title
plt.title("Feature importances")

## Plot a graph using all of the normalized features
plt.bar(range(Z.shape[1]), importances[indices],
       color="b", yerr=std[indices], align="center")

plt.xticks(range(Z.shape[1]), spam_feature_cols[indices], rotation=90)
plt.xlim([-1, Z.shape[1]])

## Show the graph!
plt.show()
```

```
Feature ranking:
1. feature 51 (0.141203)
2. feature 23 (0.076135)
3. feature 6 (0.073537)
4. feature 52 (0.073031)
5. feature 54 (0.061571)
6. feature 55 (0.061143)
7. feature 56 (0.058942)
8. feature 15 (0.049227)
9. feature 20 (0.047997)
10. feature 22 (0.032252)
```


Feature importances
0.25
0.20
0.15
0.10
0.05
0.00
char_freq_!
word_freq_money
word_freq_remove
char_freq_$
capital_run_length_average
capital_run_length_longest
capital_run_length_total
word_freq_free
word_freq_your
word_freq_000
word_freq_hp
word_freq_1999
word_freq_our
word_freq_you
word_freq_all
char_freq_(
word_freq_business
word_freq_over
word_freq_george
word_freq_receive
word_freq_hpl
word_freq_edu
word_freq_email
word_freq_will
word_freq_re
word_freq_mail
word_freq_internet
word_freq_650
word_freq_meeting
char_freq_#
word_freq_address
word_freq_make
word_freq_people
word_freq_technology
word_freq_order
char_freq_;
word_freq_credit
word_freq_font
word_freq_85
char_freq_[
word_freq_pm
word_freq_report
word_freq_labs
word_freq_original
word_freq_addresses
word_freq_data
word_freq_project
word_freq_direct
word_freq_conference
word_freq_parts
word_freq_415
word_freq_cs
word_freq_lab
word_freq_telnet
word_freq_3d
word_freq_857
word_freq_table

Similar to the analysis above, we don't see much need for many of these features; maybe only the first thirty or so will be important for performing this classification task.

One final feature analysis, using a technique known as step-forward feature selection.

Step forward feature selection starts with the evaluation of each individual feature, and selects that which results in the best performing selected algorithm model. What's the "best?" That depends entirely on the defined evaluation criteria (AUC, prediction accuracy, RMSE, etc.). I am using accuracy as this is a classification task.

Next, all possible combinations of the that selected feature and a subsequent feature are evaluated, and a second feature is selected, and so on, until the required predefined number of features is selected.

In [268]:
```
## Build RF classifier to use in feature selection
clf = ensemble.RandomForestClassifier()

# Build step forward feature selection
sfs1 = sfs(clf,
           k_features=6,
           forward=True, # Otherwise, this will be the backward selection
           floating=False,
           n_jobs=10, # The number of CPUs to use for evaluating
           verbose=2,
           scoring='accuracy',
           cv=5)

# Perform SFFS
sfs1 = sfs1.fit(X_train_standard, y_train)
```

```
[Parallel(n_jobs=10)]: Using backend LokyBackend with 10 concurrent workers.
[Parallel(n_jobs=10)]: Done  21 tasks      | elapsed:    4.9s
[Parallel(n_jobs=10)]: Done  57 out of  57 | elapsed:    5.3s finished

[2019-10-31 16:08:16] Features: 1/6 -- score: 0.7779549202525267[Parallel(n_jobs=10)]: Using backend LokyBackend with
10 concurrent workers.
[Parallel(n_jobs=10)]: Done  21 tasks      | elapsed:    2.5s
[Parallel(n_jobs=10)]: Done  56 out of  56 | elapsed:    3.0s finished

[2019-10-31 16:08:20] Features: 2/6 -- score: 0.8388206859921853[Parallel(n_jobs=10)]: Using backend LokyBackend with
10 concurrent workers.
[Parallel(n_jobs=10)]: Done  21 tasks      | elapsed:    2.4s
[Parallel(n_jobs=10)]: Done  55 out of  55 | elapsed:    2.9s finished

[2019-10-31 16:08:23] Features: 3/6 -- score: 0.8680132610309345[Parallel(n_jobs=10)]: Using backend LokyBackend with
10 concurrent workers.
[Parallel(n_jobs=10)]: Done  21 tasks      | elapsed:    2.5s
[Parallel(n_jobs=10)]: Done  54 out of  54 | elapsed:    3.1s finished

[2019-10-31 16:08:26] Features: 4/6 -- score: 0.8903754533969794[Parallel(n_jobs=10)]: Using backend LokyBackend with
10 concurrent workers.
[Parallel(n_jobs=10)]: Done  21 tasks      | elapsed:    2.4s
[Parallel(n_jobs=10)]: Done  53 out of  53 | elapsed:    2.9s finished

[2019-10-31 16:08:30] Features: 5/6 -- score: 0.9040371742279195[Parallel(n_jobs=10)]: Using backend LokyBackend with
10 concurrent workers.
[Parallel(n_jobs=10)]: Done  21 tasks      | elapsed:    2.4s
[Parallel(n_jobs=10)]: Done  52 out of  52 | elapsed:    2.9s finished

[2019-10-31 16:08:33] Features: 6/6 -- score: 0.9155259375009127
```

In [287]:
```
## Now we print out what feature columns of Linear Regressor picked out - we can use these to subset later
## We'll remember to grab these columns further along in the analysis
feat_cols = list(sfs1.k_feature_idx_)
print(feat_cols)
```

```
[6, 24, 26, 45, 52, 55]
```

In [204]:
```
## Display the column names related to our "premier" features
for i in spambase_df.columns[feat_cols]:
    print(i)
```

```
word_freq_remove
word_freq_hp
word_freq_george
word_freq_edu
char_freq_$
capital_run_length_longest
```

Based on this analysis, the top six features are presented above.

In [14]:
```
## Create new scaling object

sc = StandardScaler()

## Standardize dataset

X_train_standard = sc.fit_transform(X_train)
X_test_standard = sc.transform(X_test)
```

B. Model Creation and Evaluation

1. Create parameter grids for each model
2. Used nested cross validation to determine the best model
3. Tune the hyper parameters for the best model
4. Evaluate the models on the testing data

Here are the different models I will be using for my analysis for this classification problem

1. Logistic(linear) Classifier
2. k-Nearest Neighbors Classifier
3. Decision Tree Classifier
4. SVM Classifier
5. Ensemble/Gradient Boost CLassifier
6. Neural Network (Keras & KerasTuner)

In [135]:

```
######### SETTING UP PARAMETER GRIDS FOR THE DIFFERENT DATA MINING MODELS WE ARE GOING TO USE ###########

## Set up a grid for the Logit Classifier
lc_p_grid = {"penalty": ["l2"],
            "C": [1, 5, 10, 50, 1000],
            "solver": ["liblinear"]}

## Set up a grid for kNN Classifier
## Going to use 1-30 neighbors, and two different distance calculations
knnc_p_grid = {"n_neighbors": list(range(1, 31)),
             "weights": ["uniform", "distance"]}

## Set up a grid for the DecisionTree Classifier
dtc_p_grid = {"criterion": ["gini"],
            "splitter": ["best"],
            "max_features": [15, 20, 25],
            "max_depth": [5, 10, 15]}

## Set up a grid for the Support Vector Classifier
svc_p_grid = {"C": [1, 10, 50, 1000],
              "gamma": [0.0001, 0.0005, 0.001, 0.005],
             "kernel": ["poly", "rbf"]}

## Set up a grid for GBoost Classifier
gbc_p_grid = {'loss': ["deviance"],
                  'n_estimators': [100, 200, 300, 400, 500],
                  'max_depth': [3, 4, 5],
                  'min_samples_split': [2, 4, 6],
                  'max_features': [5, 10],
                  'criterion': ["mse"],
                  'learning_rate': [0.01]}
```

In [136]:
```
## Set a number of trials to run for the models

num_trials = 20

## Empty arrays to store scores for classifier
nested_scores_lc = np.zeros(num_trials)
nested_scores_knnc = np.zeros(num_trials)
nested_scores_dtc = np.zeros(num_trials)
nested_scores_svc = np.zeros(num_trials)
nested_scores_gbc = np.zeros(num_trials)
```

In [137]:
```
## Create new regressors for each data mining technique

## Linear Regression
lc = linear_model.LogisticRegression()

## k-Nearest Neighbors
knnc = neighbors.KNeighborsClassifier()

## Decision Tree
dtc = tree.DecisionTreeClassifier()

## Support Vector Machine
svc = svm.SVC()

## Gradient Boost
gbc = ensemble.GradientBoostingClassifier()
```

In [303]:
```
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    lclass= GridSearchCV(estimator=lc, param_grid=lc_p_grid, cv=inner_cv)
    lclass.fit(X_train_standard, y_train)

    nested_score = cross_val_score(lclass, X = X_train_standard, y = y_train, cv = outer_cv)
    nested_scores_lc[i] = nested_score.mean()
```

In [305]:
```python
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    knnclass= GridSearchCV(estimator=knnc, param_grid=knnc_p_grid, cv=inner_cv)
    knnclass.fit(X_train_standard, y_train)

    nested_score = cross_val_score(knnclass, X = X_train_standard, y = y_train, cv = outer_cv)
    nested_scores_knnr[i] = nested_score.mean()
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-305-16a7d48e722e> in <module>
     12     knnclass.fit(X_train_standard, y_train)
     13 
---> 14     nested_score = cross_val_score(knnclass, X = X_train_standard, y = y_train, cv = outer_cv)
     15     nested_scores_knnr[i] = nested_score.mean()

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in cross_val_score(estimator, X, y, groups, scorin
g, cv, n_jobs, verbose, fit_params, pre_dispatch, error_score)
    400                                 fit_params=fit_params,
    401                                 pre_dispatch=pre_dispatch,
--> 402                                 error_score=error_score)
    403     return cv_results['test_score']
    404 

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in cross_validate(estimator, X, y, groups, scoring,
cv, n_jobs, verbose, fit_params, pre_dispatch, return_train_score, return_estimator, error_score)
    238             return_times=True, return_estimator=return_estimator,
    239             error_score=error_score)
--> 240         for train, test in cv.split(X, y, groups))
    241 
    242     zipped_scores = list(zip(*scores))

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    915             # remaining jobs.
    916             self._iterating = False
--> 917             if self.dispatch_one_batch(iterator):
    918                 self._iterating = self._original_iterator is not None
    919 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
```

```
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550
    551     def get(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _fit_and_score(estimator, X, y, scorer, train, t
est, verbose, parameters, fit_params, return_train_score, return_parameters, return_n_test_samples, return_times, retu
rn_estimator, error_score)
    526             estimator.fit(X_train, **fit_params)
    527         else:
--> 528             estimator.fit(X_train, y_train, **fit_params)
    529
    530     except Exception as e:

C:\Python\lib\site-packages\sklearn\model_selection\_search.py in fit(self, X, y, groups, **fit_params)
    720                 return results_container[0]
    721
--> 722             self._run_search(evaluate_candidates)
    723
    724         results = results_container[0]

C:\Python\lib\site-packages\sklearn\model_selection\_search.py in _run_search(self, evaluate_candidates)
   1189     def _run_search(self, evaluate_candidates):
   1190         """Search all candidates in param_grid"""
-> 1191         evaluate_candidates(ParameterGrid(self.param_grid))
   1192
   1193
```

```
C:\Python\lib\site-packages\sklearn\model_selection\_search.py in evaluate_candidates(candidate_params)
    709                                for parameters, (train, test)
    710                                in product(candidate_params,
--> 711                                           cv.split(X, y, groups)))
    712 
    713                 all_candidate_params.extend(candidate_params)

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    918                 self._iterating = self._original_iterator is not None
    919 
--> 920             while self.dispatch_one_batch(iterator):
    921                 pass
    922 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550 
    551     def get(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
```

```
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _fit_and_score(estimator, X, y, scorer, train, test, verbose, parameters, fit_params, return_train_score, return_parameters, return_n_test_samples, return_times, return_estimator, error_score)
    570         if return_train_score:
    571             train_scores = _score(estimator, X_train, y_train, scorer,
--> 572                                   is_multimetric)
    573
    574     if verbose > 2:

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _score(estimator, X_test, y_test, scorer, is_multimetric)
    603     """
    604     if is_multimetric:
--> 605         return _multimetric_score(estimator, X_test, y_test, scorer)
    606     else:
    607         if y_test is None:

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _multimetric_score(estimator, X_test, y_test, scorers)
    633             score = scorer(estimator, X_test)
    634         else:
--> 635             score = scorer(estimator, X_test, y_test)
    636
    637         if hasattr(score, 'item'):

C:\Python\lib\site-packages\sklearn\metrics\scorer.py in _passthrough_scorer(estimator, *args, **kwargs)
    239 def _passthrough_scorer(estimator, *args, **kwargs):
    240     """Function that wraps estimator.score"""
--> 241     return estimator.score(*args, **kwargs)
    242
    243

C:\Python\lib\site-packages\sklearn\base.py in score(self, X, y, sample_weight)
    288         """
    289         from .metrics import accuracy_score
--> 290         return accuracy_score(y, self.predict(X), sample_weight=sample_weight)
    291
    292
```

```
C:\Python\lib\site-packages\sklearn\neighbors\classification.py in predict(self, X)
    147         X = check_array(X, accept_sparse='csr')
    148 
--> 149         neigh_dist, neigh_ind = self.kneighbors(X)
    150         classes_ = self.classes_
    151         _y = self._y

C:\Python\lib\site-packages\sklearn\neighbors\base.py in kneighbors(self, X, n_neighbors, return_distance)
    453                 delayed_query(
    454                     self._tree, X[s], n_neighbors, return_distance)
--> 455                 for s in gen_even_slices(X.shape[0], n_jobs)
    456             )
    457         else:

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    915             # remaining jobs.
    916             self._iterating = False
--> 917             if self.dispatch_one_batch(iterator):
    918                 self._iterating = self._original_iterator is not None
    919 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550 
    551     def get(self):
```

```
C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\neighbors\base.py in _tree_query_parallel_helper(tree, data, n_neighbors, return_d
istance)
    290     under PyPy.
    291     """
--> 292     return tree.query(data, n_neighbors, return_distance)
    293 
    294 

KeyboardInterrupt:
```

In [318]:

```
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    dtclass= GridSearchCV(estimator=dtc, param_grid=dtc_p_grid, cv=inner_cv)
    dtclass.fit(X_train_standard, y_train)

    nested_score = cross_val_score(dtreg, X = X_train_standard, y = y_train, cv = outer_cv)
    nested_scores_dtc[i] = nested_score.mean()
```

In [138]:

```
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Logit Regression

    svclass= GridSearchCV(estimator=svc, param_grid=svc_p_grid, cv=inner_cv)
    svclass.fit(X_train_standard, y_train)

    nested_score = cross_val_score(svclass, X = X_train_standard, y = y_train, cv = outer_cv)
    nested_scores_svc[i] = nested_score.mean()
```

In [365]:
```
## Loop for each trial
for i in range(20):

    ## Choose cross-validation techniques for the inner and outer loops,
    ## independently of the dataset.
    inner_cv = KFold(n_splits = 4, shuffle = True, random_state = i)
    outer_cv = KFold(n_splits = 4, shuffle = True, random_state = i)

## Nested CV for Gradient Boost Classifier

    gbclass= GridSearchCV(estimator=gbc, param_grid=gbc_p_grid, cv=inner_cv)
    gbclass.fit(X_train_standard, y_train)

    nested_score = cross_val_score(gbclass, X = X_train_standard, y = y_train, cv = outer_cv)
    nested_scores_gbc[i] = nested_score.mean()
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-365-60868a14b3d3> in <module>
     12     gbclass.fit(X_train_standard, y_train)
     13 
---> 14     nested_score = cross_val_score(gbclass, X = X_train_standard, y = y_train, cv = outer_cv)
     15     nested_scores_gbc[i] = nested_score.mean()

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in cross_val_score(estimator, X, y, groups, scorin
g, cv, n_jobs, verbose, fit_params, pre_dispatch, error_score)
    400                                 fit_params=fit_params,
    401                                 pre_dispatch=pre_dispatch,
--> 402                                 error_score=error_score)
    403     return cv_results['test_score']
    404 

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in cross_validate(estimator, X, y, groups, scoring,
cv, n_jobs, verbose, fit_params, pre_dispatch, return_train_score, return_estimator, error_score)
    238             return_times=True, return_estimator=return_estimator,
    239             error_score=error_score)
--> 240         for train, test in cv.split(X, y, groups))
    241 
    242     zipped_scores = list(zip(*scores))

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    918                 self._iterating = self._original_iterator is not None
    919 
--> 920             while self.dispatch_one_batch(iterator):
    921                 pass
    922 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
```

```
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550 
    551     def get(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _fit_and_score(estimator, X, y, scorer, train, t
est, verbose, parameters, fit_params, return_train_score, return_parameters, return_n_test_samples, return_times, retu
rn_estimator, error_score)
    526             estimator.fit(X_train, **fit_params)
    527         else:
--> 528             estimator.fit(X_train, y_train, **fit_params)
    529 
    530     except Exception as e:

C:\Python\lib\site-packages\sklearn\model_selection\_search.py in fit(self, X, y, groups, **fit_params)
    720                 return results_container[0]
    721 
--> 722             self._run_search(evaluate_candidates)
    723 
    724         results = results_container[0]

C:\Python\lib\site-packages\sklearn\model_selection\_search.py in _run_search(self, evaluate_candidates)
   1189     def _run_search(self, evaluate_candidates):
   1190         """Search all candidates in param_grid"""
-> 1191         evaluate_candidates(ParameterGrid(self.param_grid))
   1192 
   1193 
```

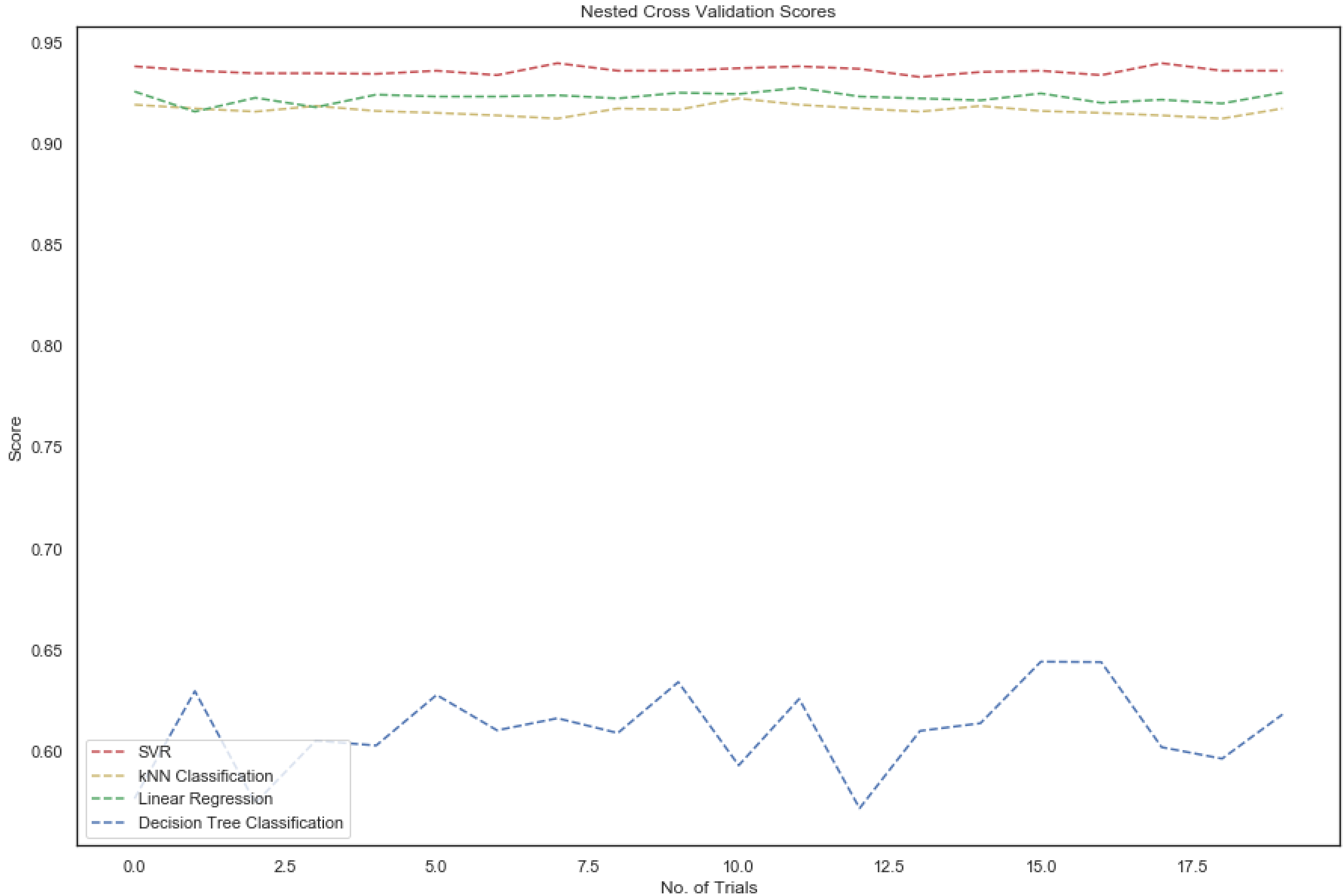
```
C:\Python\lib\site-packages\sklearn\model_selection\_search.py in evaluate_candidates(candidate_params)
    709                                for parameters, (train, test)
    710                                in product(candidate_params,
--> 711                                           cv.split(X, y, groups)))
    712 
    713                 all_candidate_params.extend(candidate_params)

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self, iterable)
    918                 self._iterating = self._original_iterator is not None
    919 
--> 920             while self.dispatch_one_batch(iterator):
    921                 pass
    922 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in dispatch_one_batch(self, iterator)
    757                 return False
    758             else:
--> 759                 self._dispatch(tasks)
    760                 return True
    761 

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in _dispatch(self, batch)
    714         with self._lock:
    715             job_idx = len(self._jobs)
--> 716             job = self._backend.apply_async(batch, callback=cb)
    717             # A job can complete so quickly than its callback is
    718             # called before we get here, causing self._jobs to

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in apply_async(self, func, callback)
    180     def apply_async(self, func, callback=None):
    181         """Schedule a func to be run"""
--> 182         result = ImmediateResult(func)
    183         if callback:
    184             callback(result)

C:\Python\lib\site-packages\sklearn\externals\joblib\_parallel_backends.py in __init__(self, batch)
    547         # Don't delay the application, to avoid keeping the input
    548         # arguments in memory
--> 549         self.results = batch()
    550 
    551     def get(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in __call__(self)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226 
```

```
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\externals\joblib\parallel.py in <listcomp>(.0)
    223         with parallel_backend(self._backend, n_jobs=self._n_jobs):
    224             return [func(*args, **kwargs)
--> 225                     for func, args, kwargs in self.items]
    226
    227     def __len__(self):

C:\Python\lib\site-packages\sklearn\model_selection\_validation.py in _fit_and_score(estimator, X, y, scorer, train, t
est, verbose, parameters, fit_params, return_train_score, return_parameters, return_n_test_samples, return_times, retu
rn_estimator, error_score)
    526             estimator.fit(X_train, **fit_params)
    527         else:
--> 528             estimator.fit(X_train, y_train, **fit_params)
    529
    530     except Exception as e:

C:\Python\lib\site-packages\sklearn\ensemble\gradient_boosting.py in fit(self, X, y, sample_weight, monitor)
   1463         n_stages = self._fit_stages(X, y, y_pred, sample_weight, self._rng,
   1464                                     X_val, y_val, sample_weight_val,
-> 1465                                     begin_at_stage, monitor, X_idx_sorted)
   1466
   1467         # change shape of arrays after fit (early-stopping or additional ests)

C:\Python\lib\site-packages\sklearn\ensemble\gradient_boosting.py in _fit_stages(self, X, y, y_pred, sample_weight, ra
ndom_state, X_val, y_val, sample_weight_val, begin_at_stage, monitor, X_idx_sorted)
   1527             y_pred = self._fit_stage(i, X, y, y_pred, sample_weight,
   1528                                      sample_mask, random_state, X_idx_sorted,
-> 1529                                      X_csc, X_csr)
   1530
   1531             # track deviance (= loss)

C:\Python\lib\site-packages\sklearn\ensemble\gradient_boosting.py in _fit_stage(self, i, X, y, y_pred, sample_weight,
 sample_mask, random_state, X_idx_sorted, X_csc, X_csr)
   1192             X = X_csr if X_csr is not None else X
   1193             tree.fit(X, residual, sample_weight=sample_weight,
-> 1194                      check_input=False, X_idx_sorted=X_idx_sorted)
   1195
   1196             # update tree leaves

C:\Python\lib\site-packages\sklearn\tree\tree.py in fit(self, X, y, sample_weight, check_input, X_idx_sorted)
   1140             sample_weight=sample_weight,
   1141             check_input=check_input,
-> 1142             X_idx_sorted=X_idx_sorted)
   1143         return self
   1144
```

```
C:\Python\lib\site-packages\sklearn\tree\tree.py in fit(self, X, y, sample_weight, check_input, X_idx_sorted)
    364                                             min_impurity_split)
    365
--> 366         builder.build(self.tree_, X, y, sample_weight, X_idx_sorted)
    367
    368         if self.n_outputs_ == 1:

KeyboardInterrupt:
```

In [356]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot nested scores for each classifier - quickly visual the best performing model
## This is WITHOUT having changed any of the default parameters
plt.plot(nested_scores_svc, 'r--', label = "SVR")
plt.plot(nested_scores_knnc, 'y--', label = "kNN Classification")
plt.plot(nested_scores_lc, 'g--', label = "Linear Regression")
plt.plot(nested_scores_dtc, 'b--', label = "Decision Tree Classification")
plt.plot(nested_scores_gbc, 'v--', label = "Gradient Boost")


## Give some labels
plt.xlabel("No. of Trials")
plt.ylabel("Score")

## Title and Legend
plt.title("Nested Cross Validation Scores")
plt.legend(loc = 'lower left')

## Show the graph
plt.show()
```



From our Cross Validation plot above, we see that a Support Vector Machine classifier performs the best at classifying spam email out of the different data mining procedures tested.

Since it is pretty clear that this was an effective model, I will do further exploration using the best parameters selected by the nested cross validation process.

In [140]:
```
## Get our best params and their scores
print(svclass.best_params_)
print()

## Print out how well it performed using the best params
print(svclass.best_score_)

## save our best params so we can use them in our actual SVC model!
best_svc_params = svclass.best_params_
```

```
{'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}

0.9347826086956522
```

In [147]:
```
## Initialize our DecisionTree classifier with the best params based on our GridSearch.
svc_clf = svm.SVC(**best_svc_params, probability = True)

## Train the model (fit the data)
# 'fit' builds a decision tree from the training set (X, y).
svc_clf = svc_clf.fit(X_train_standard, y_train)
```

In [148]:
```
## Evaluate performance by cross-validation
scores = cross_val_score(svc_clf, X_train_standard, y_train, cv = 10)
print(scores)

print()

# The mean score and the 95% confidence interval of our scores:
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std() * 2))
```

```
[0.92569659 0.91925466 0.91925466 0.94099379 0.93167702 0.92546584
 0.93478261 0.94099379 0.95341615 0.93457944]

Accuracy: 0.93 (+/- 0.02)
```

In [149]:
```
## Create three empty lists to store my precision, recall, and average precision scores
precision, recall, average_precision = [], [], []

## Get probabilities for our labels
probas_ = svc_clf.predict_proba(X_test_standard)
```

In [156]:

```python
## Create my false positive rate, true positive rate, and threshold using my test data
fpr, tpr, threholds = roc_curve(y_test, probas_[:, 1], pos_label = 1)

## Create precision and recall scores to plot with
precision_score, recall_score, _ = precision_recall_curve(y_test, probas_[:, 1])

## Calculate the overall AUC for the model
auc = np.trapz(tpr, fpr)

## Save the average precision for our model
avg_precision = average_precision_score(y_test, probas_[:, 1])
```

In [206]:

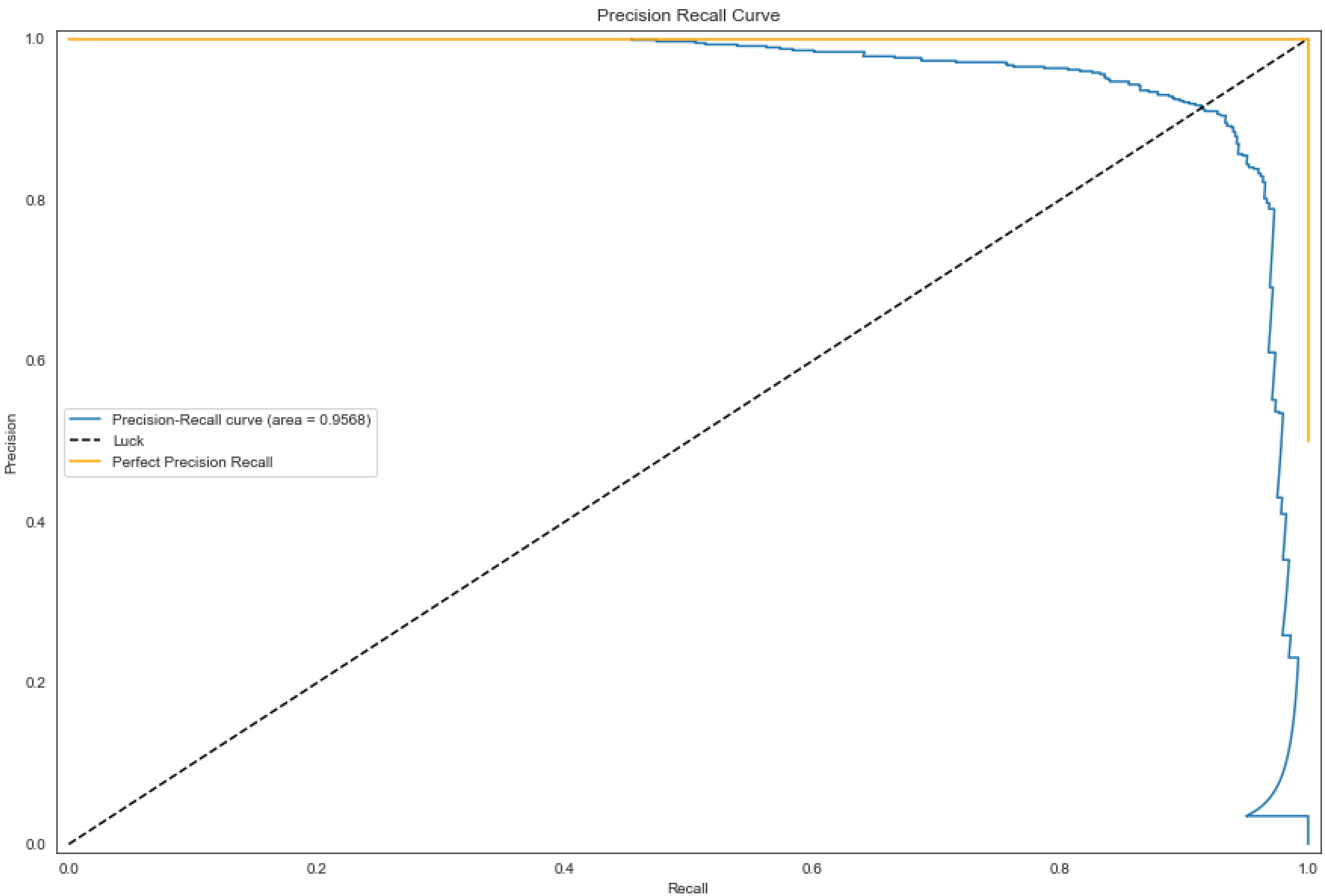
```python
## Build a confusion matrix from our Support Vector Machine model - another assessment of performance

## Try to predict the outcomes on our test data
predicted = svc_clf.predict(X_test_standard)

## Compare that with our ACTUAL values from the test data set
matrix = confusion_matrix(y_test, predicted)
print(matrix)
print()

## Create a report to show our precision(accuracy), recall, and f1 for predictions
report = classification_report(y_test, predicted)
print(report)
```

```
[[798  39]
 [ 51 493]]

              precision    recall  f1-score   support

           0       0.94      0.95      0.95       837
           1       0.93      0.91      0.92       544

    accuracy                           0.93      1381
   macro avg       0.93      0.93      0.93      1381
weighted avg       0.93      0.93      0.93      1381
```

The AUC Curve is very, very close to being an almost perfect line. This tells me this model is very effective at discovering and identifying spam emails.

In [198]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Create a new figure to plot
plt.figure(figsize= (15, 10))

lw = 2


## Plot the model's Precision Recall Curve
plt.plot(precision_score, recall_score, linestyle = '-', label='Precision-Recall curve (area = {})'.format(round(avg_pr
ecision, 4)))

## Put in a line to demonstrate blind luck
plt.plot([0, 1], [0, 1], color = 'black', linestyle = '--', label = 'Luck')

## Plot the perfect Precision Recall Line to see how well our SVM Classifier performed
plt.plot([0, 1], [1, 1], color = 'orange')
plt.plot([1, 1], [1, 0.5], color = 'orange', label = "Perfect Precision Recall")

## Set the limits so they start at zero
plt.ylim([-0.01, 1.01])
plt.xlim([-0.01, 1.01])

## Set labels for x and y
plt.xlabel('Recall')
plt.ylabel('Precision')

## Set a title and legend
plt.title('Precision Recall Curve')
plt.legend(loc = 'center left')

## Show the curve!
plt.show()
```


Precision Recall Curve
Precision-Recall curve (area = 0.9568)
Luck
Perfect Precision Recall
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0

Due to the slight class imbalance, I also generate a Precision-Recall curve; accuracy may not be an effective measure of model performance so it is good to just confirm the model is working as expected.

Here we confirm our earlier observation that the model is performing extremely well, with an area of 0.9568.

To reinforce and showcase how the model is performing at its classification task, I include two lines: one showing random luck (black) and a perfect Precision-Recall (orange).

I also want to investigate the cost for misclassification; it's not enough to make sure the model is working as expected; we want to penalize the model for mistakes (not too harshly) to make sure that we don't allow e-mails through that should not be allowed in.

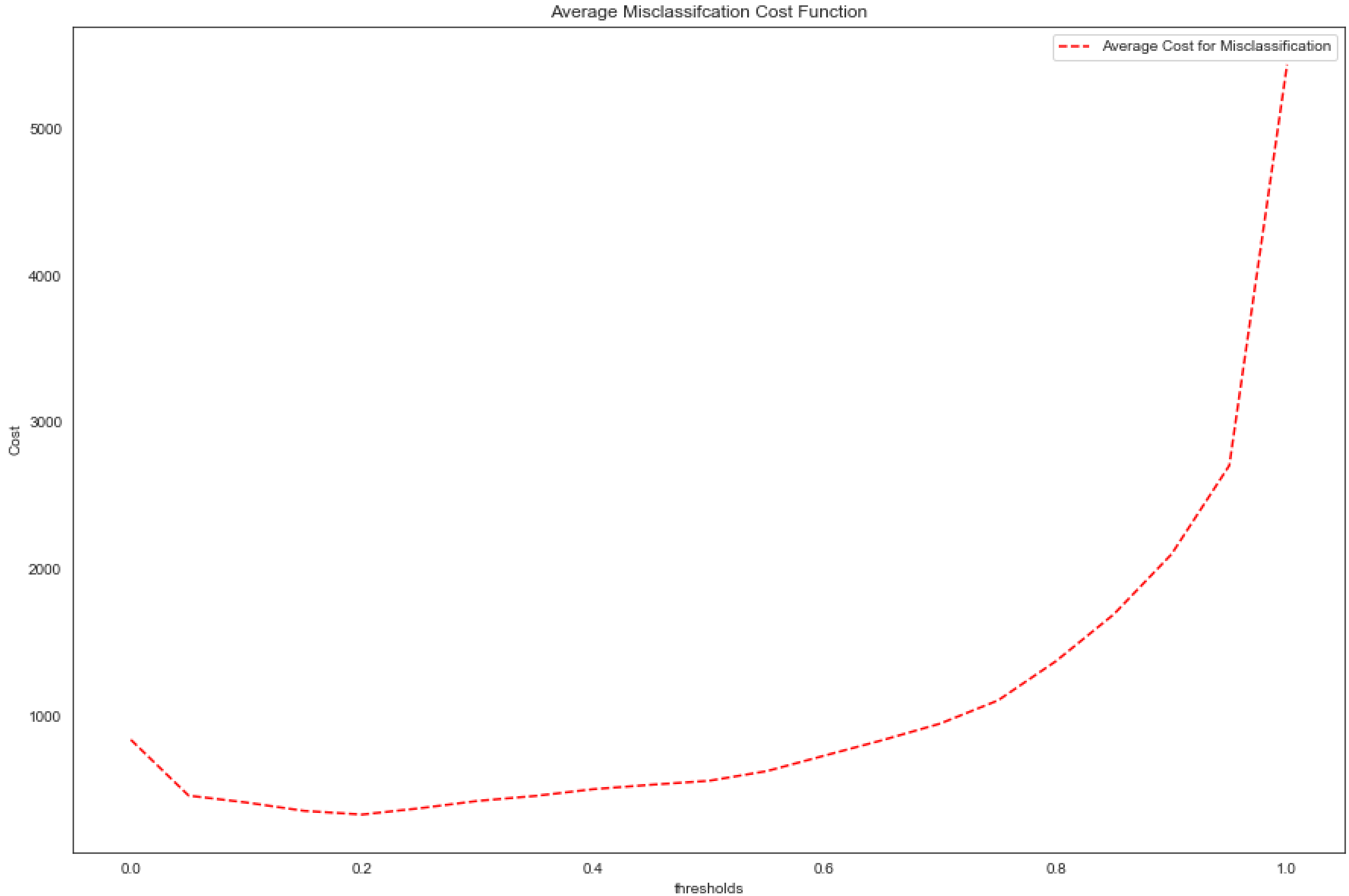
I will punish false positives more harshly than false negative. The reasoning behind this is if an email that *should* have been identified as spam get through, then it should be penalized vs. if the classifier misclassified a good email as spam.

If a spam email gets through the filter, then any harmful software associated with that spam message may infect the user's computer down the line.

In [ ]:
```
## Initialize different thresholds and Costs that will be tied to those thresholds
thresholds = np.linspace(0, 1.0, num=21)

## Generate three generic Cost List for each matrix
Cost_List = np.linspace(0, 1.0, num=21)

## Build cost matrix for misidentifying spam
cost_matrix = np.array([[0, 1], [10, 0]])
```

In [199]:

```
index = 0

for t in thresholds:

    predict_thre = np.where(probas_[:, 1] > t, 1, 0)  ## prediction based on the preset threshold
    clf_matrix = confusion_matrix(y_test, predict_thre)
    Cost_List[index] = clf_matrix[0][0]*cost_matrix[0][0] +clf_matrix[0][1]*cost_matrix[0][1] +clf_matrix[1][0]*cost_ma
trix[1][0] +clf_matrix[1][1]*cost_matrix[1][1]
    index+=1

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot each Cost Line individually
plt.plot(thresholds, Cost_List, 'r--', label = "Average Cost for Misclassification")

## Give some labels
plt.xlabel("thresholds")
plt.ylabel("Cost")

## Title and Legend
plt.title("Average Misclassifcation Cost Function")
plt.legend(loc = 'upper right')

## Show the Cost Matrix Analysis
plt.show()
```



C. Use Keras and Keras Tuner to develop some classifiers

1. Build the model function
2. Build the hyper tuner function
3. Tune the model and explore the data space to make predictions
4. Assess the model performance.

```
In [15]: ## Just confirming the shape of the training data to know how many features we have
X_train_standard.shape
```

```
Out[15]: (3220, 57)
```

Now that I can use Keras and hyper tuning, I will create a classifier using a similar process.

```
In [16]: ## First, we build a function to actually put together our model

def build_class_model(hp):

    ## Base layer
    cmodel = Sequential()

    ## Add first layer, test with 10 up to 22 features
    cmodel.add(layers.Dense(units = hp.Int ("units",
                                            min_value = 20,
                                            max_value = 57,
                                            step = 2),
                            ## Use same initializer as model above
                            activation = "relu", kernel_initializer = "he_uniform"))

    ## Add target layer using linear regressor
    cmodel.add(layers.Dense(1, activation = "sigmoid"))

    ## Use different learning rates to test the model
    cmodel.compile(
    optimizer = keras.optimizers.SGD(
        hp.Choice("learning_rate", values = [1e-2, 1e-3, 1e-4])),


    ## Use same loss fucntion as before
    loss = "binary_crossentropy",

    ## Test using Accuracy

    metrics = ["accuracy", "binary_accuracy"])

    ## Return completed model
    return cmodel
```

In [99]:

```
## Build hyper tuner, will perform cross validation for us

### Use RandomSearch

classtuner = RandomSearch(

    ## Use model function from above
    build_class_model,

    ## What is objective function?  Using loss here
    objective = "val_accuracy",

    ## Set number of trials
    max_trials = 5,

    ## Number of executions
    executions_per_trial = 3,

    ## Set to short dir path
    directory = "C:\\"

)
```

In [22]:
```
classtuner.search(X_train_standard, y_train, epochs = 10,
                  validation_data = (X_test_standard, y_test))
```

```
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 1:15 - loss: 0.6932 - accuracy: 0.6250 - binary_accuracy: 0.62 - ET
A: 2s - loss: 0.8565 - accuracy: 0.4414 - binary_accuracy: 0.4414 - ETA: 1s - loss: 0.8521 - accuracy: 0.4428 - binary
_accuracy: 0.44 - ETA: 0s - loss: 0.8269 - accuracy: 0.4636 - binary_accuracy: 0.46 - ETA: 0s - loss: 0.8334 - accurac
y: 0.4611 - binary_accuracy: 0.46 - ETA: 0s - loss: 0.8260 - accuracy: 0.4605 - binary_accuracy: 0.46 - 1s 400us/sampl
e - loss: 0.8271 - accuracy: 0.4584 - binary_accuracy: 0.4584 - val_loss: 0.8325 - val_accuracy: 0.4881 - val_binary_a
ccuracy: 0.4881
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.8710 - accuracy: 0.4375 - binary_accuracy: 0.43 - ETA:
0s - loss: 0.8556 - accuracy: 0.4315 - binary_accuracy: 0.43 - ETA: 0s - loss: 0.8306 - accuracy: 0.4617 - binary_accu
racy: 0.46 - ETA: 0s - loss: 0.8270 - accuracy: 0.4587 - binary_accuracy: 0.45 - ETA: 0s - loss: 0.8212 - accuracy: 0.
4604 - binary_accuracy: 0.46 - ETA: 0s - loss: 0.8184 - accuracy: 0.4633 - binary_accuracy: 0.46 - 0s 123us/sample - l
oss: 0.8215 - accuracy: 0.4637 - binary_accuracy: 0.4637 - val_loss: 0.8270 - val_accuracy: 0.4902 - val_binary_accura
cy: 0.4902
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 1.0493 - accuracy: 0.4062 - binary_accuracy: 0.40 - ETA:
0s - loss: 0.8164 - accuracy: 0.4492 - binary_accuracy: 0.44 - ETA: 0s - loss: 0.8136 - accuracy: 0.4614 - binary_accu
racy: 0.46 - ETA: 0s - loss: 0.8119 - accuracy: 0.4701 - binary_accuracy: 0.47 - ETA: 0s - loss: 0.8165 - accuracy: 0.
4719 - binary_accuracy: 0.47 - 0s 108us/sample - loss: 0.8159 - accuracy: 0.4699 - binary_accuracy: 0.4699 - val_loss:
0.8217 - val_accuracy: 0.4953 - val_binary_accuracy: 0.4953
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.9919 - accuracy: 0.3438 - binary_accuracy: 0.34 - ETA:
0s - loss: 0.8072 - accuracy: 0.4688 - binary_accuracy: 0.46 - ETA: 0s - loss: 0.8172 - accuracy: 0.4744 - binary_accu
racy: 0.47 - ETA: 0s - loss: 0.8118 - accuracy: 0.4750 - binary_accuracy: 0.47 - ETA: 0s - loss: 0.8132 - accuracy: 0.
4770 - binary_accuracy: 0.47 - ETA: 0s - loss: 0.8116 - accuracy: 0.4782 - binary_accuracy: 0.47 - 0s 122us/sample - l
oss: 0.8104 - accuracy: 0.4783 - binary_accuracy: 0.4783 - val_loss: 0.8164 - val_accuracy: 0.5018 - val_binary_accura
cy: 0.5018
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 1.0000 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.8319 - accuracy: 0.4851 - binary_accuracy: 0.48 - ETA: 0s - loss: 0.8214 - accuracy: 0.4805 - binary_accu
racy: 0.48 - ETA: 0s - loss: 0.8073 - accuracy: 0.4916 - binary_accuracy: 0.49 - ETA: 0s - loss: 0.8012 - accuracy: 0.
4823 - binary_accuracy: 0.48 - ETA: 0s - loss: 0.8040 - accuracy: 0.4807 - binary_accuracy: 0.48 - 0s 155us/sample - l
oss: 0.8050 - accuracy: 0.4832 - binary_accuracy: 0.4832 - val_loss: 0.8112 - val_accuracy: 0.5040 - val_binary_accura
cy: 0.5040
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7132 - accuracy: 0.5312 - binary_accuracy: 0.53 - ETA:
0s - loss: 0.8309 - accuracy: 0.4458 - binary_accuracy: 0.44 - ETA: 0s - loss: 0.8025 - accuracy: 0.4706 - binary_accu
racy: 0.47 - ETA: 0s - loss: 0.7988 - accuracy: 0.4778 - binary_accuracy: 0.47 - ETA: 0s - loss: 0.7930 - accuracy: 0.
4860 - binary_accuracy: 0.48 - ETA: 0s - loss: 0.7976 - accuracy: 0.4840 - binary_accuracy: 0.48 - ETA: 0s - loss: 0.7
997 - accuracy: 0.4842 - binary_accuracy: 0.48 - 1s 160us/sample - loss: 0.7997 - accuracy: 0.4870 - binary_accuracy:
0.4870 - val_loss: 0.8061 - val_accuracy: 0.5083 - val_binary_accuracy: 0.5083
Epoch 7/10
1312/3220 [===========>..................] - ETA: 0s - loss: 0.8565 - accuracy: 0.4375 - binary_accuracy: 0.43 - ETA:
0s - loss: 0.8094 - accuracy: 0.4700 - binary_accuracy: 0.47 - ETA: 0s - loss: 0.8189 - accuracy: 0.4710 - binary_accu
racy: 0.47
```

limit_output extension: Maximum message size of 10000 exceeded with 10070 characters

In [23]:
```
best_models = classtuner.get_best_models(num_models = 2)
```

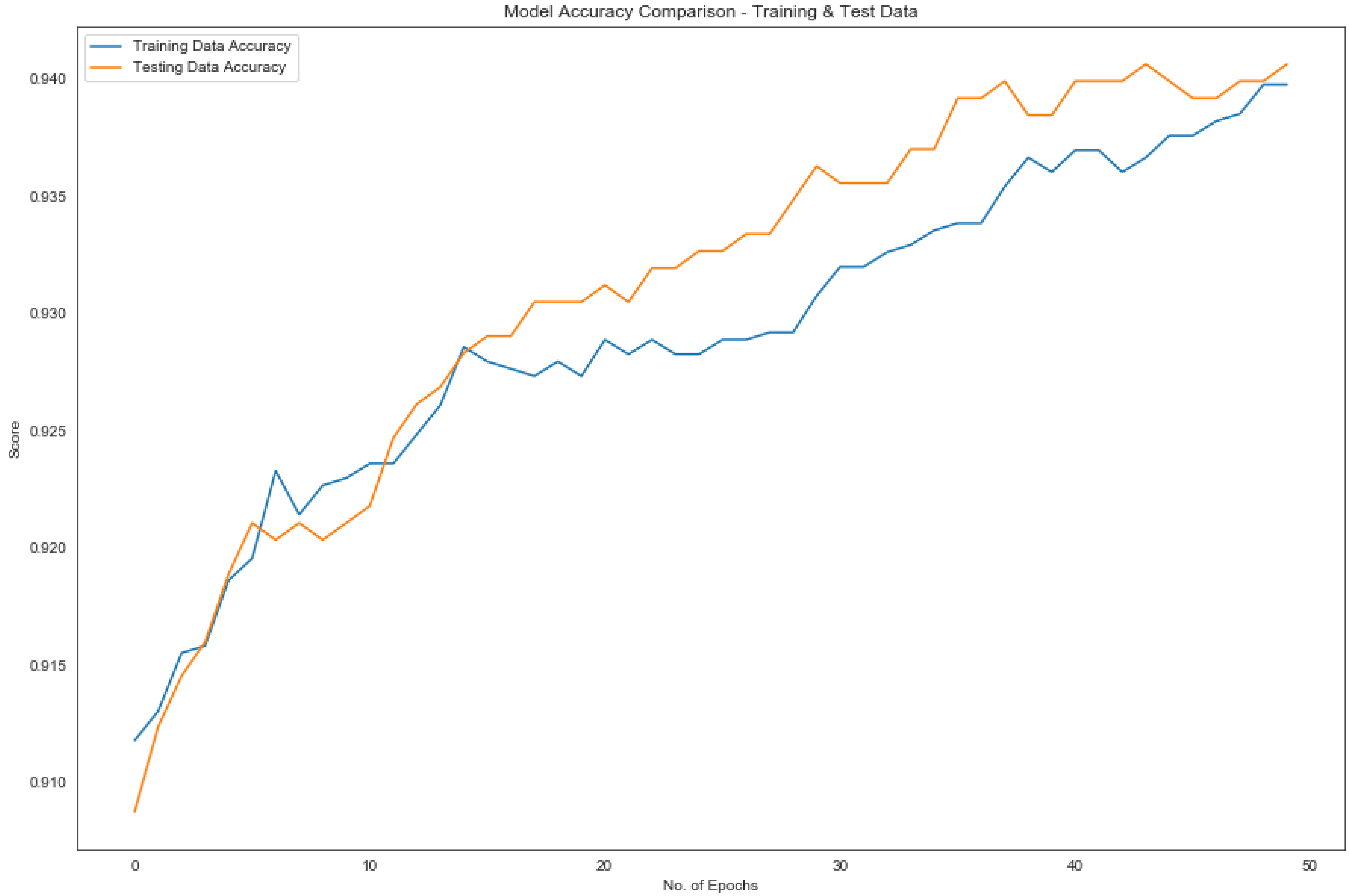
In [24]:
```
top_model = best_models[0]
```

In [25]:
```
### Use our best model and reverify it against our test data

spam_history = top_model.fit(X_train_standard, y_train,
                validation_data = (X_test_standard, y_test), epochs = 50, verbose = 0)
```

In [26]:
```
train_acc = top_model.evaluate(X_train_standard, y_train, verbose=0)
test_acc = top_model.evaluate(X_test_standard, y_test, verbose=0)
```

In [29]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot nested scores for each classifier - quickly visual the best performing model
## This is WITHOUT having changed any of the default parameters
plt.plot(spam_history.history['accuracy'], label = "Training Data Accuracy")
plt.plot(spam_history.history['val_accuracy'], label = "Testing Data Accuracy")

## Give some labels and title
plt.xlabel("No. of Epochs")
plt.ylabel("Score")

## Title and Legend
plt.title("Model Accuracy Comparison - Training & Test Data")
plt.legend()

## Show the graph
plt.show()
```

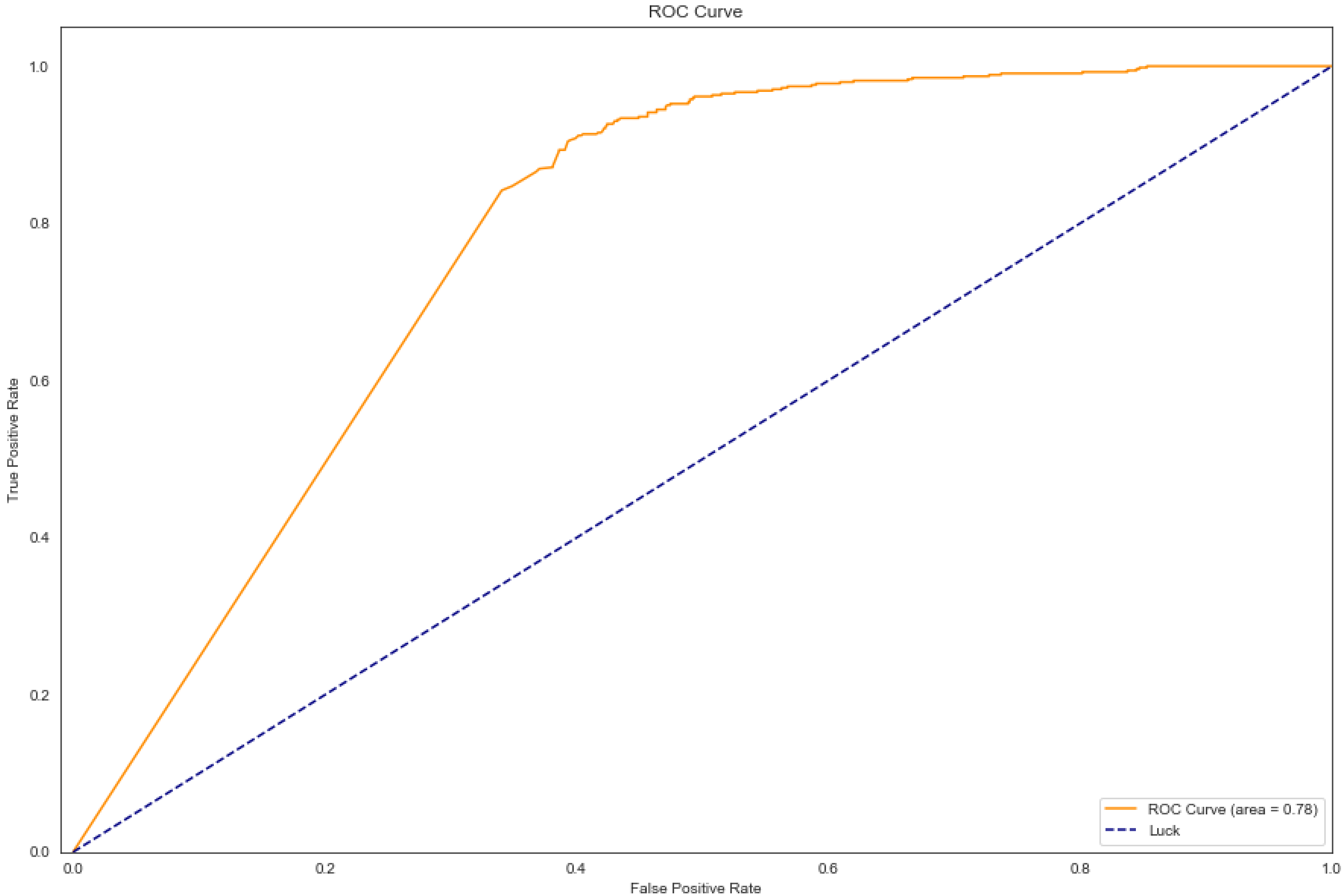


Using `sklearn` metrics library, I can determine how well the model is performing by generating ROC, Precision, and Recall curves

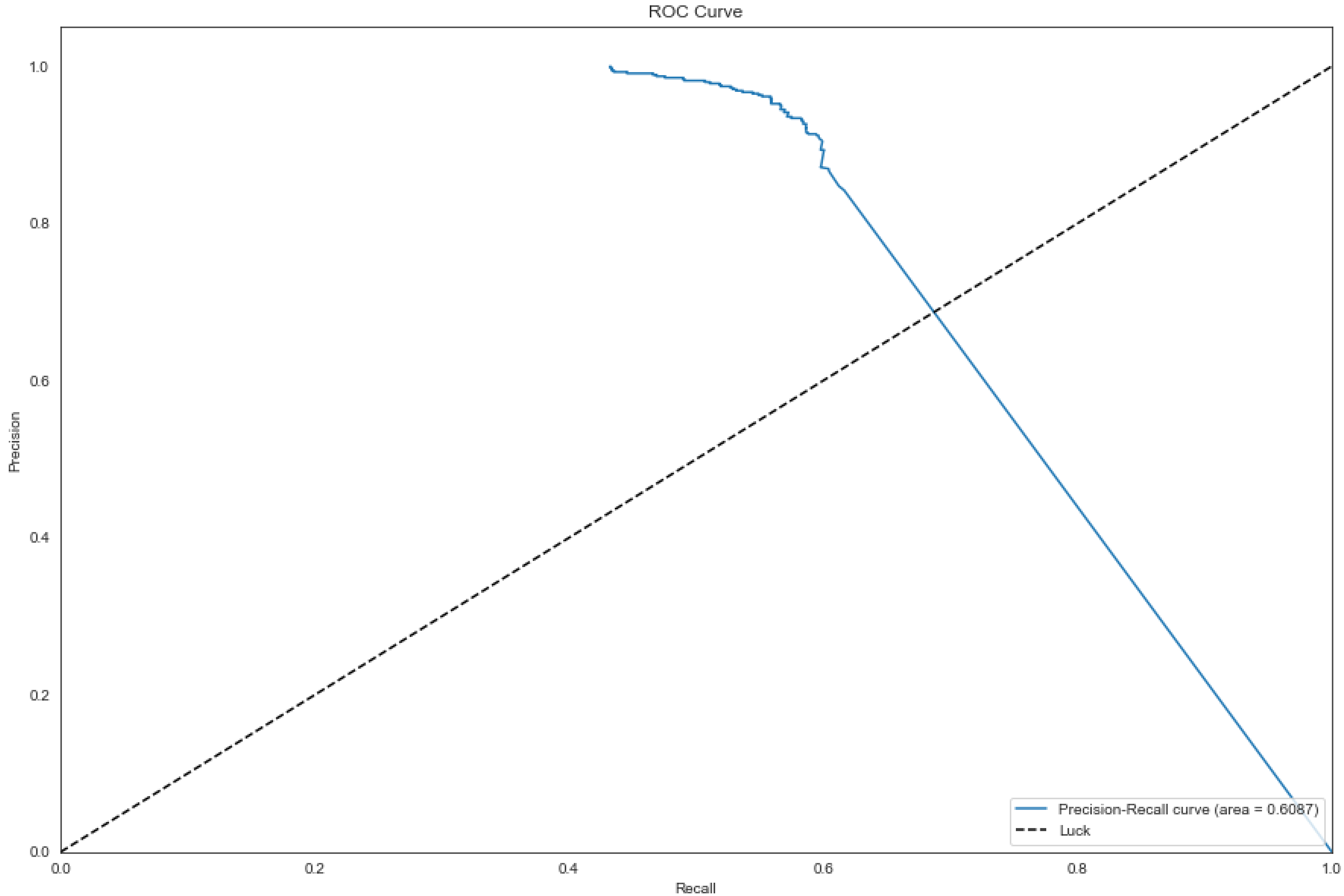
In [30]:
```
## Create three empty lists to store my precision, recall, and average precision scores
precision, recall, average_precision = [], [], []

## Get probabilities for our labels
probas_ = top_model.predict_proba(X_test)
```

In [69]:
```
## Create my false positive rate, true positive rate, and threshold using my test data
fpr, tpr, threholds = roc_curve(y_test, probas_, pos_label = 1)

## Create precision and recall scores to plot with
precision_score, recall_score, _ = precision_recall_curve(y_test, probas_)

## Calculate the overall AUC for the model
auc = np.trapz(tpr, fpr)

## Save the average precision for our model
avg_precision = average_precision_score(y_test, probas_)
```

In [41]:
```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Create a new figure to plot
plt.figure(figsize= (15, 10))

lw = 2

## Draw the line for my fpr and tpr
plt.plot(fpr, tpr, color = 'darkorange',
        label = 'ROC Curve (area = %0.2f)' % auc)

## Put in a line to demonstrate blind luck
plt.plot([0, 1], [0, 1], color = 'navy', linestyle = '--', label = 'Luck')

## Set the limits of the plot for better visualization
plt.xlim([-0.01, 1.0])
plt.ylim([0.0, 1.05])

## Set labels for x and y
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

## Set a title and legend
plt.title('ROC Curve')
plt.legend(loc = 'lower right')

## Show the curve!
plt.show()
```


ROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC Curve (area = 0.78)
Luck

In [77]:
```python
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Create a new figure to plot
plt.figure(figsize= (15, 10))

lw = 2


## Plot the model's Precision Recall Curve
plt.plot(precision_score, recall_score, label='Precision-Recall curve (area = {})'.format(round(avg_precision, 4)))

## Put in a line to demonstrate blind luck
plt.plot([0, 1], [0, 1], color = 'black', linestyle = '--', label = 'Luck')

## Set the limits so they start at zero
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])

## Set labels for x and y
plt.xlabel('Recall')
plt.ylabel('Precision')

## Set a title and legend
plt.title('ROC Curve')
plt.legend(loc = 'lower right')

## Show the curve!
plt.show()
```



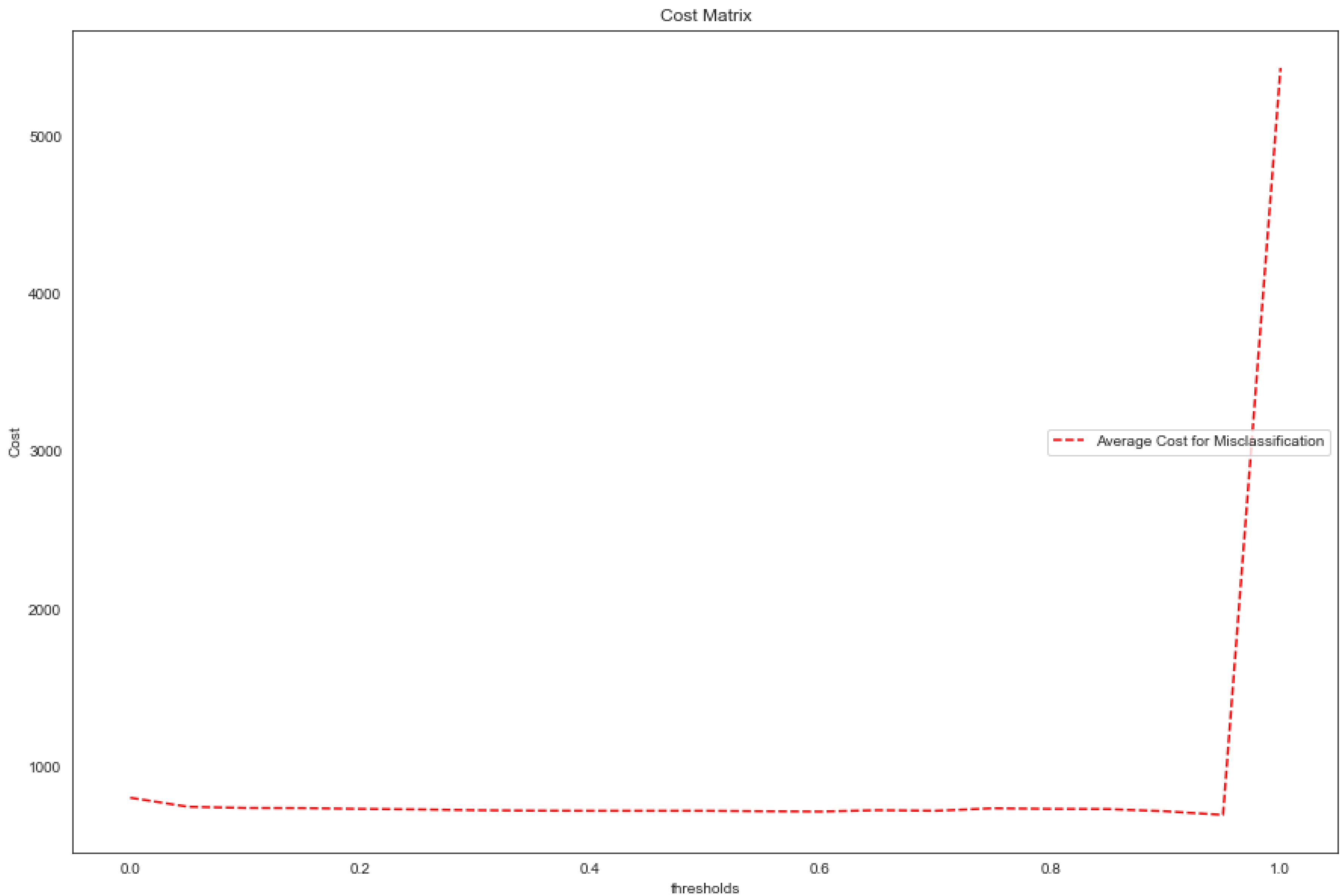
I will now be building a cost matrix and visualization for this; again this is to visualize and assess how the model is "punishing" mistakes.

In [61]:
```
## Initialize different thresholds and Costs that will be tied to those thresholds
thresholds = np.linspace(0, 1.0, num=21)

## Generate three generic Cost List for each matrix
Cost_List = np.linspace(0, 1.0, num=21)

## Build cost matrix for misidentifying spam
cost_matrix = np.array([[0, 1], [10, 0]])
```

In [64]:
```
## Show the cost matrix
cost_matrix
```

Out[64]:
```
array([[ 0,  1],
       [10,  0]])
```

In [62]:
```
## Set the index to zero

index = 0

### Loop through our different thresholds and calculate the misclassification cost at each interval
for t in thresholds:

    predict_thre = np.where(probas_ > t, 1, 0)  ## prediction based on the preset threshold
    clf_matrix = confusion_matrix(y_test, predict_thre)
    Cost_List[index] = clf_matrix[0][0]*cost_matrix[0][0] +clf_matrix[0][1]*cost_matrix[0][1] +clf_matrix[1][0]*cost_ma
trix[1][0] +clf_matrix[1][1]*cost_matrix[1][1]
    index+=1

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot each Cost Line individually
plt.plot(thresholds, Cost_List, 'r--', label = "Average Cost for Misclassification")

## Give some labels
plt.xlabel("thresholds")
plt.ylabel("Cost")

## Title and Legend
plt.title("Cost Matrix")
plt.legend(loc = 'right')

## Show the Cost Matrix Analysis
plt.show()
```



I am going to try one more model, using a Linear activator instead. We'll see if we can improve the accuracy of the model.

In [120]:
```
## First, we build a function to actually put together our model - with change to use a linear activation instead

def build_class_model(hp):

    ## Base layer
    cmodel = Sequential()

    ## Add first layer, test with 30 up to 100 layers
    cmodel.add(layers.Dense(units = hp.Int ("units",
                                              min_value = 30,
                                              max_value = 100,
                                              step = 2),
                           ## Use same initializer as model above
                           activation = "linear", kernel_initializer = "he_uniform"))

    ## Add target layer using linear regressor
    cmodel.add(layers.Dense(1, activation = "linear"))

    ## Use different learning rates to test the model
    cmodel.compile(
    optimizer = keras.optimizers.SGD(
        hp.Choice("learning_rate", values = [1e-2, 1e-3, 1e-4])),


    ## Use same loss fucntion as before
    loss = "binary_crossentropy",

    ## Test using Accuracy

    metrics = ["accuracy", "binary_accuracy"])

    ## Return completed model
    return cmodel
```

In [121]:
```
## Build hyper tuner, will perform cross validation for us

### Use RandomSearch

classtuner = RandomSearch(

    ## Use model function from above
    build_class_model,

    ## What is objective function?  Using loss here
    objective = "val_accuracy",

    ## Set number of trials
    max_trials = 5,

    ## Number of executions
    executions_per_trial = 3,

    ## Set to short dir path
    directory = "C:\\"

)
```

In [122]:

```
classtuner.search(X_train_standard, y_train, epochs = 10,
                  validation_data = (X_test_standard, y_test))
```

```
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 41s - loss: 0.6667 - accuracy: 0.5938 - binary_accuracy: 0.593 - ET
A: 1s - loss: 0.6533 - accuracy: 0.5960 - binary_accuracy: 0.596 - ETA: 0s - loss: 0.6518 - accuracy: 0.6094 - binary_
accuracy: 0.60 - ETA: 0s - loss: 0.6526 - accuracy: 0.6068 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6480 - accurac
y: 0.6157 - binary_accuracy: 0.61 - 1s 266us/sample - loss: 0.6494 - accuracy: 0.6140 - binary_accuracy: 0.6140 - val_
loss: 0.6507 - val_accuracy: 0.6119 - val_binary_accuracy: 0.6119
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7399 - accuracy: 0.4688 - binary_accuracy: 0.46 - ETA:
0s - loss: 0.6469 - accuracy: 0.6175 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6442 - accuracy: 0.6263 - binary_accu
racy: 0.62 - ETA: 0s - loss: 0.6481 - accuracy: 0.6174 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6484 - accuracy: 0.
6159 - binary_accuracy: 0.61 - 0s 100us/sample - loss: 0.6489 - accuracy: 0.6146 - binary_accuracy: 0.6146 - val_loss:
0.6502 - val_accuracy: 0.6133 - val_binary_accuracy: 0.6133
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6465 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6462 - accuracy: 0.6116 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6496 - accuracy: 0.6079 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6456 - accuracy: 0.6197 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6484 - accuracy: 0.
6144 - binary_accuracy: 0.61 - 0s 96us/sample - loss: 0.6485 - accuracy: 0.6146 - binary_accuracy: 0.6146 - val_loss:
0.6498 - val_accuracy: 0.6140 - val_binary_accuracy: 0.6140
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6453 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6439 - accuracy: 0.6262 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6454 - accuracy: 0.6256 - binary_accu
racy: 0.62 - ETA: 0s - loss: 0.6476 - accuracy: 0.6192 - binary_accuracy: 0.61 - 0s 91us/sample - loss: 0.6480 - accur
acy: 0.6143 - binary_accuracy: 0.6143 - val_loss: 0.6494 - val_accuracy: 0.6148 - val_binary_accuracy: 0.6148
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5964 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6447 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6472 - accuracy: 0.6152 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6465 - accuracy: 0.6170 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6463 - accuracy: 0.
6170 - binary_accuracy: 0.61 - 0s 92us/sample - loss: 0.6476 - accuracy: 0.6137 - binary_accuracy: 0.6137 - val_loss:
0.6489 - val_accuracy: 0.6140 - val_binary_accuracy: 0.6140
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6227 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6404 - accuracy: 0.6377 - binary_accuracy: 0.63 - ETA: 0s - loss: 0.6496 - accuracy: 0.6138 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6491 - accuracy: 0.6125 - binary_accuracy: 0.61 - 0s 84us/sample - loss: 0.6472 - accur
acy: 0.6140 - binary_accuracy: 0.6140 - val_loss: 0.6485 - val_accuracy: 0.6140 - val_binary_accuracy: 0.6140
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6752 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6449 - accuracy: 0.6224 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6468 - accuracy: 0.6100 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6450 - accuracy: 0.6187 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6474 - accuracy: 0.
6138 - binary_accuracy: 0.61 - 0s 92us/sample - loss: 0.6467 - accuracy: 0.6146 - binary_accuracy: 0.6146 - val_loss:
0.6481 - val_accuracy: 0.6148 - val_binary_accuracy: 0.6148
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5999 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6559 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6480 - accuracy: 0.6114 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6449 - accuracy: 0.6187 - binary_accuracy: 0.61 - 0s 84us/sample - loss: 0.6463 - accur
acy: 0.6152 - binary_accuracy: 0.6152 - val_loss: 0.6477 - val_accuracy: 0.6148 - val_binary_accuracy: 0.6148
Epoch 9/10
```

```
3220/3220 [==============================] - ETA: 0s - loss: 0.5928 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6475 - accuracy: 0.6274 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6421 - accuracy: 0.6284 - binary_accu
racy: 0.62 - ETA: 0s - loss: 0.6429 - accuracy: 0.6258 - binary_accuracy: 0.62 - 0s 92us/sample - loss: 0.6458 - accur
acy: 0.6149 - binary_accuracy: 0.6149 - val_loss: 0.6473 - val_accuracy: 0.6155 - val_binary_accuracy: 0.6155
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6615 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6335 - accuracy: 0.6525 - binary_accuracy: 0.65 - ETA: 0s - loss: 0.6417 - accuracy: 0.6289 - binary_accu
racy: 0.62 - ETA: 0s - loss: 0.6438 - accuracy: 0.6206 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6455 - accuracy: 0.
6160 - binary_accuracy: 0.61 - 0s 99us/sample - loss: 0.6454 - accuracy: 0.6149 - binary_accuracy: 0.6149 - val_loss:
0.6468 - val_accuracy: 0.6162 - val_binary_accuracy: 0.6162
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 46s - loss: 0.7933 - accuracy: 0.5000 - binary_accuracy: 0.500 - ET
A: 1s - loss: 0.6857 - accuracy: 0.6150 - binary_accuracy: 0.615 - ETA: 0s - loss: 0.6858 - accuracy: 0.6165 - binary_
accuracy: 0.61 - ETA: 0s - loss: 0.6926 - accuracy: 0.6071 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6952 - accurac
y: 0.6001 - binary_accuracy: 0.60 - 1s 268us/sample - loss: 0.6949 - accuracy: 0.6009 - binary_accuracy: 0.6009 - val_
loss: 0.6929 - val_accuracy: 0.6025 - val_binary_accuracy: 0.6025
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7374 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 0.7084 - accuracy: 0.5775 - binary_accuracy: 0.57 - ETA: 0s - loss: 0.6935 - accuracy: 0.5987 - binary_accu
racy: 0.59 - ETA: 0s - loss: 0.6908 - accuracy: 0.6042 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6947 - accuracy: 0.
5988 - binary_accuracy: 0.59 - 0s 108us/sample - loss: 0.6938 - accuracy: 0.6003 - binary_accuracy: 0.6003 - val_loss:
0.6919 - val_accuracy: 0.6032 - val_binary_accuracy: 0.6032
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6361 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.7066 - accuracy: 0.5872 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6988 - accuracy: 0.5943 - binary_accu
racy: 0.59 - ETA: 0s - loss: 0.6906 - accuracy: 0.6014 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6918 - accuracy: 0.
6016 - binary_accuracy: 0.60 - 0s 92us/sample - loss: 0.6928 - accuracy: 0.6000 - binary_accuracy: 0.6000 - val_loss:
0.6909 - val_accuracy: 0.6032 - val_binary_accuracy: 0.6032
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7541 - accuracy: 0.5312 - binary_accuracy: 0.53 - ETA:
0s - loss: 0.6831 - accuracy: 0.6131 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6864 - accuracy: 0.6062 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6881 - accuracy: 0.6072 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6927 - accuracy: 0.
6001 - binary_accuracy: 0.60 - 0s 99us/sample - loss: 0.6918 - accuracy: 0.6003 - binary_accuracy: 0.6003 - val_loss:
0.6899 - val_accuracy: 0.6017 - val_binary_accuracy: 0.6017
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6411 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6631 - accuracy: 0.6288 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6756 - accuracy: 0.6170 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6810 - accuracy: 0.6105 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6903 - accuracy: 0.
6003 - binary_accuracy: 0.60 - 0s 95us/sample - loss: 0.6908 - accuracy: 0.5997 - binary_accuracy: 0.5997 - val_loss:
0.6889 - val_accuracy: 0.6010 - val_binary_accuracy: 0.6010
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7421 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6894 - accuracy: 0.5981 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6877 - accuracy: 0.6023 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6827 - accuracy: 0.6079 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6896 - accuracy: 0.
5993 - binary_accuracy: 0.59 - 0s 93us/sample - loss: 0.6898 - accuracy: 0.6000 - binary_accuracy: 0.6000 - val_loss:
0.6880 - val_accuracy: 0.6010 - val_binary_accuracy: 0.6010
Epoch 7/10
```

```
3220/3220 [==============================] - ETA: 0s - loss: 0.6851 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6980 - accuracy: 0.5925 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6895 - accuracy: 0.6012 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6876 - accuracy: 0.6039 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6898 - accuracy: 0.
5981 - binary_accuracy: 0.59 - 0s 88us/sample - loss: 0.6888 - accuracy: 0.5997 - binary_accuracy: 0.5997 - val_loss:
0.6871 - val_accuracy: 0.6003 - val_binary_accuracy: 0.6003
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7333 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6812 - accuracy: 0.6080 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6862 - accuracy: 0.6050 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6896 - accuracy: 0.5977 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6876 - accuracy: 0.
6000 - binary_accuracy: 0.60 - 0s 89us/sample - loss: 0.6879 - accuracy: 0.5994 - binary_accuracy: 0.5994 - val_loss:
0.6862 - val_accuracy: 0.6003 - val_binary_accuracy: 0.6003
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6300 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6676 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6791 - accuracy: 0.6111 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6843 - accuracy: 0.6028 - binary_accuracy: 0.60 - 0s 81us/sample - loss: 0.6870 - accur
acy: 0.5997 - binary_accuracy: 0.5997 - val_loss: 0.6853 - val_accuracy: 0.6003 - val_binary_accuracy: 0.6003
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6740 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6673 - accuracy: 0.6115 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6759 - accuracy: 0.6062 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6797 - accuracy: 0.6029 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6851 - accuracy: 0.
5992 - binary_accuracy: 0.59 - 0s 99us/sample - loss: 0.6861 - accuracy: 0.5988 - binary_accuracy: 0.5988 - val_loss:
0.6845 - val_accuracy: 0.6003 - val_binary_accuracy: 0.6003
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 51s - loss: 1.3191 - accuracy: 0.4062 - binary_accuracy: 0.406 - ET
A: 1s - loss: 1.3215 - accuracy: 0.4152 - binary_accuracy: 0.415 - ETA: 0s - loss: 1.3374 - accuracy: 0.4042 - binary_
accuracy: 0.40 - ETA: 0s - loss: 1.3485 - accuracy: 0.3962 - binary_accuracy: 0.39 - 1s 285us/sample - loss: 1.3541 -
accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.3440 - val_accuracy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 1.5227 - accuracy: 0.3125 - binary_accuracy: 0.31 - ETA:
0s - loss: 1.2931 - accuracy: 0.4139 - binary_accuracy: 0.41 - ETA: 0s - loss: 1.3312 - accuracy: 0.3940 - binary_accu
racy: 0.39 - 0s 55us/sample - loss: 1.3299 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.3201 - val_accur
acy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 1.4284 - accuracy: 0.3438 - binary_accuracy: 0.34 - ETA:
0s - loss: 1.2620 - accuracy: 0.4231 - binary_accuracy: 0.42 - ETA: 0s - loss: 1.3065 - accuracy: 0.3944 - binary_accu
racy: 0.39 - ETA: 0s - loss: 1.3089 - accuracy: 0.3927 - binary_accuracy: 0.39 - 0s 82us/sample - loss: 1.3062 - accur
acy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.2967 - val_accuracy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 1.2505 - accuracy: 0.4062 - binary_accuracy: 0.40 - ETA:
0s - loss: 1.2633 - accuracy: 0.4090 - binary_accuracy: 0.40 - ETA: 0s - loss: 1.2964 - accuracy: 0.3923 - binary_accu
racy: 0.39 - 0s 65us/sample - loss: 1.2832 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.2739 - val_accur
acy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.9407 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 1.3229 - accuracy: 0.3663 - binary_accuracy: 0.36 - ETA: 0s - loss: 1.2737 - accuracy: 0.3883 - binary_accu
racy: 0.38 - 0s 58us/sample - loss: 1.2606 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.2517 - val_accur
acy: 0.3939 - val_binary_accuracy: 0.3939
```

```
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 1.4851 - accuracy: 0.2812 - binary_accuracy: 0.28 - ETA:
0s - loss: 1.2588 - accuracy: 0.3906 - binary_accuracy: 0.39 - ETA: 0s - loss: 1.2517 - accuracy: 0.3877 - binary_accu
racy: 0.38 - 0s 54us/sample - loss: 1.2387 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.2300 - val_accur
acy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 1.0662 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 1.2263 - accuracy: 0.3956 - binary_accuracy: 0.39 - ETA: 0s - loss: 1.2244 - accuracy: 0.3946 - binary_accu
racy: 0.39 - ETA: 0s - loss: 1.2266 - accuracy: 0.3909 - binary_accuracy: 0.39 - ETA: 0s - loss: 1.2257 - accuracy: 0.
3904 - binary_accuracy: 0.39 - ETA: 0s - loss: 1.2160 - accuracy: 0.3960 - binary_accuracy: 0.39 - 0s 120us/sample - l
oss: 1.2174 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.2089 - val_accuracy: 0.3939 - val_binary_accura
cy: 0.3939
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 1.3568 - accuracy: 0.3438 - binary_accuracy: 0.34 - ETA:
0s - loss: 1.1989 - accuracy: 0.3940 - binary_accuracy: 0.39 - ETA: 0s - loss: 1.2141 - accuracy: 0.3862 - binary_accu
racy: 0.38 - ETA: 0s - loss: 1.1992 - accuracy: 0.3931 - binary_accuracy: 0.39 - 0s 86us/sample - loss: 1.1966 - accur
acy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.1884 - val_accuracy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 1.2619 - accuracy: 0.3125 - binary_accuracy: 0.31 - ETA:
0s - loss: 1.1704 - accuracy: 0.4004 - binary_accuracy: 0.40 - ETA: 0s - loss: 1.1905 - accuracy: 0.3878 - binary_accu
racy: 0.38 - 0s 56us/sample - loss: 1.1764 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.1684 - val_accur
acy: 0.3939 - val_binary_accuracy: 0.3939
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.9287 - accuracy: 0.5312 - binary_accuracy: 0.53 - ETA:
0s - loss: 1.1727 - accuracy: 0.3917 - binary_accuracy: 0.39 - ETA: 0s - loss: 1.1539 - accuracy: 0.3982 - binary_accu
racy: 0.39 - 0s 54us/sample - loss: 1.1567 - accuracy: 0.3941 - binary_accuracy: 0.3941 - val_loss: 1.1491 - val_accur
acy: 0.3939 - val_binary_accuracy: 0.3939
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.0001

|-units: 36

|-Score: 0.5377745032310486

|-Best step: 0

```
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 1:03 - loss: 0.7872 - accuracy: 0.3750 - binary_accuracy: 0.37 - ET
A: 1s - loss: 0.7291 - accuracy: 0.4256 - binary_accuracy: 0.4256 - ETA: 0s - loss: 0.7011 - accuracy: 0.5022 - binary
_accuracy: 0.50 - ETA: 0s - loss: 0.6844 - accuracy: 0.5471 - binary_accuracy: 0.54 - 1s 328us/sample - loss: 0.6797 -
accuracy: 0.5593 - binary_accuracy: 0.5593 - val_loss: 0.6360 - val_accuracy: 0.6510 - val_binary_accuracy: 0.6510
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6154 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6256 - accuracy: 0.6643 - binary_accuracy: 0.66 - ETA: 0s - loss: 0.6201 - accuracy: 0.6668 - binary_accu
racy: 0.66 - 0s 75us/sample - loss: 0.6181 - accuracy: 0.6674 - binary_accuracy: 0.6674 - val_loss: 0.6000 - val_accur
acy: 0.6843 - val_binary_accuracy: 0.6843
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6506 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.5961 - accuracy: 0.7018 - binary_accuracy: 0.70 - ETA: 0s - loss: 0.5907 - accuracy: 0.7024 - binary_accu
racy: 0.70 - ETA: 0s - loss: 0.5860 - accuracy: 0.7070 - binary_accuracy: 0.70 - 0s 81us/sample - loss: 0.5851 - accur
acy: 0.7090 - binary_accuracy: 0.7090 - val_loss: 0.5689 - val_accuracy: 0.7234 - val_binary_accuracy: 0.7234
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4946 - accuracy: 0.8125 - binary_accuracy: 0.81 - ETA:
0s - loss: 0.5595 - accuracy: 0.7393 - binary_accuracy: 0.73 - ETA: 0s - loss: 0.5562 - accuracy: 0.7447 - binary_accu
racy: 0.74 - 0s 62us/sample - loss: 0.5550 - accuracy: 0.7472 - binary_accuracy: 0.7472 - val_loss: 0.5401 - val_accur
acy: 0.7610 - val_binary_accuracy: 0.7610
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5856 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.5284 - accuracy: 0.7897 - binary_accuracy: 0.78 - ETA: 0s - loss: 0.5273 - accuracy: 0.7824 - binary_accu
racy: 0.78 - 0s 65us/sample - loss: 0.5272 - accuracy: 0.7835 - binary_accuracy: 0.7835 - val_loss: 0.5131 - val_accur
acy: 0.7907 - val_binary_accuracy: 0.7907
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5006 - accuracy: 0.7812 - binary_accuracy: 0.78 - ETA:
0s - loss: 0.5141 - accuracy: 0.8003 - binary_accuracy: 0.80 - ETA: 0s - loss: 0.5032 - accuracy: 0.8069 - binary_accu
racy: 0.80 - 0s 62us/sample - loss: 0.5012 - accuracy: 0.8106 - binary_accuracy: 0.8106 - val_loss: 0.4881 - val_accur
acy: 0.8219 - val_binary_accuracy: 0.8219
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4646 - accuracy: 0.8125 - binary_accuracy: 0.81 - ETA:
0s - loss: 0.4827 - accuracy: 0.8326 - binary_accuracy: 0.83 - ETA: 0s - loss: 0.4749 - accuracy: 0.8355 - binary_accu
racy: 0.83 - 0s 61us/sample - loss: 0.4771 - accuracy: 0.8326 - binary_accuracy: 0.8326 - val_loss: 0.4649 - val_accur
acy: 0.8371 - val_binary_accuracy: 0.8371
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4548 - accuracy: 0.8750 - binary_accuracy: 0.87 - ETA:
0s - loss: 0.4687 - accuracy: 0.8396 - binary_accuracy: 0.83 - ETA: 0s - loss: 0.4581 - accuracy: 0.8438 - binary_accu
racy: 0.84 - 0s 63us/sample - loss: 0.4550 - accuracy: 0.8457 - binary_accuracy: 0.8457 - val_loss: 0.4436 - val_accur
acy: 0.8487 - val_binary_accuracy: 0.8487
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4510 - accuracy: 0.8125 - binary_accuracy: 0.81 - ETA:
0s - loss: 0.4436 - accuracy: 0.8551 - binary_accuracy: 0.85 - ETA: 0s - loss: 0.4339 - accuracy: 0.8542 - binary_accu
racy: 0.85 - 0s 64us/sample - loss: 0.4348 - accuracy: 0.8522 - binary_accuracy: 0.8522 - val_loss: 0.4241 - val_accur
acy: 0.8610 - val_binary_accuracy: 0.8610
Epoch 10/10
```

```
3220/3220 [==============================] - ETA: 0s - loss: 0.3608 - accuracy: 0.9062 - binary_accuracy: 0.90 - ETA: 
0s - loss: 0.4230 - accuracy: 0.8559 - binary_accuracy: 0.85 - ETA: 0s - loss: 0.4185 - accuracy: 0.8645 - binary_accu
racy: 0.86 - 0s 64us/sample - loss: 0.4166 - accuracy: 0.8612 - binary_accuracy: 0.8612 - val_loss: 0.4065 - val_accur
acy: 0.8660 - val_binary_accuracy: 0.8660
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 38s - loss: 0.7170 - accuracy: 0.5000 - binary_accuracy: 0.500 - ET
A: 0s - loss: 0.6976 - accuracy: 0.5076 - binary_accuracy: 0.507 - ETA: 0s - loss: 0.6758 - accuracy: 0.5602 - binary_
accuracy: 0.56 - 1s 203us/sample - loss: 0.6687 - accuracy: 0.5724 - binary_accuracy: 0.5724 - val_loss: 0.6349 - val_
accuracy: 0.6365 - val_binary_accuracy: 0.6365
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6122 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA: 
0s - loss: 0.6289 - accuracy: 0.6316 - binary_accuracy: 0.63 - ETA: 0s - loss: 0.6196 - accuracy: 0.6433 - binary_accu
racy: 0.64 - 0s 60us/sample - loss: 0.6155 - accuracy: 0.6503 - binary_accuracy: 0.6503 - val_loss: 0.5970 - val_accur
acy: 0.6778 - val_binary_accuracy: 0.6778
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6435 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.5871 - accuracy: 0.6788 - binary_accuracy: 0.67 - ETA: 0s - loss: 0.5809 - accuracy: 0.6860 - binary_accu
racy: 0.68 - 0s 55us/sample - loss: 0.5798 - accuracy: 0.6904 - binary_accuracy: 0.6904 - val_loss: 0.5635 - val_accur
acy: 0.7306 - val_binary_accuracy: 0.7306
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6207 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.5596 - accuracy: 0.7287 - binary_accuracy: 0.72 - ETA: 0s - loss: 0.5503 - accuracy: 0.7444 - binary_accu
racy: 0.74 - 0s 55us/sample - loss: 0.5476 - accuracy: 0.7466 - binary_accuracy: 0.7466 - val_loss: 0.5328 - val_accur
acy: 0.7690 - val_binary_accuracy: 0.7690
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5372 - accuracy: 0.7500 - binary_accuracy: 0.75 - ETA: 
0s - loss: 0.5344 - accuracy: 0.7642 - binary_accuracy: 0.76 - ETA: 0s - loss: 0.5216 - accuracy: 0.7736 - binary_accu
racy: 0.77 - 0s 58us/sample - loss: 0.5181 - accuracy: 0.7786 - binary_accuracy: 0.7786 - val_loss: 0.5045 - val_accur
acy: 0.8009 - val_binary_accuracy: 0.8009
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4624 - accuracy: 0.8750 - binary_accuracy: 0.87 - ETA: 
0s - loss: 0.4975 - accuracy: 0.8139 - binary_accuracy: 0.81 - ETA: 0s - loss: 0.4931 - accuracy: 0.8147 - binary_accu
racy: 0.81 - 0s 53us/sample - loss: 0.4910 - accuracy: 0.8171 - binary_accuracy: 0.8171 - val_loss: 0.4785 - val_accur
acy: 0.8291 - val_binary_accuracy: 0.8291
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4806 - accuracy: 0.7500 - binary_accuracy: 0.75 - ETA: 
0s - loss: 0.4760 - accuracy: 0.8321 - binary_accuracy: 0.83 - ETA: 0s - loss: 0.4706 - accuracy: 0.8367 - binary_accu
racy: 0.83 - 0s 56us/sample - loss: 0.4661 - accuracy: 0.8373 - binary_accuracy: 0.8373 - val_loss: 0.4547 - val_accur
acy: 0.8465 - val_binary_accuracy: 0.8465
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5181 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA: 
0s - loss: 0.4535 - accuracy: 0.8408 - binary_accuracy: 0.84 - ETA: 0s - loss: 0.4484 - accuracy: 0.8445 - binary_accu
racy: 0.84 - 0s 54us/sample - loss: 0.4436 - accuracy: 0.8512 - binary_accuracy: 0.8512 - val_loss: 0.4330 - val_accur
acy: 0.8631 - val_binary_accuracy: 0.8631
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4183 - accuracy: 0.9062 - binary_accuracy: 0.90 - ETA: 
0s - loss: 0.4280 - accuracy: 0.8516 - binary_accuracy: 0.85 - ETA: 0s - loss: 0.4228 - accuracy: 0.8608 - binary_accu
```

```
racy: 0.86 - 0s 61us/sample - loss: 0.4232 - accuracy: 0.8602 - binary_accuracy: 0.8602 - val_loss: 0.4135 - val_accur
acy: 0.8812 - val_binary_accuracy: 0.8812
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4215 - accuracy: 0.8750 - binary_accuracy: 0.87 - ETA:
0s - loss: 0.4171 - accuracy: 0.8601 - binary_accuracy: 0.86 - ETA: 0s - loss: 0.4086 - accuracy: 0.8706 - binary_accu
racy: 0.87 - 0s 63us/sample - loss: 0.4048 - accuracy: 0.8714 - binary_accuracy: 0.8714 - val_loss: 0.3959 - val_accur
acy: 0.8827 - val_binary_accuracy: 0.8827
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 37s - loss: 0.6464 - accuracy: 0.5625 - binary_accuracy: 0.562 - ET
A: 0s - loss: 0.6898 - accuracy: 0.5281 - binary_accuracy: 0.528 - ETA: 0s - loss: 0.6607 - accuracy: 0.6171 - binary_
accuracy: 0.61 - 1s 201us/sample - loss: 0.6492 - accuracy: 0.6373 - binary_accuracy: 0.6373 - val_loss: 0.6057 - val_
accuracy: 0.7060 - val_binary_accuracy: 0.7060
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6236 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.6036 - accuracy: 0.7047 - binary_accuracy: 0.70 - ETA: 0s - loss: 0.5962 - accuracy: 0.7093 - binary_accu
racy: 0.70 - 0s 58us/sample - loss: 0.5921 - accuracy: 0.7127 - binary_accuracy: 0.7127 - val_loss: 0.5755 - val_accur
acy: 0.7270 - val_binary_accuracy: 0.7270
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5431 - accuracy: 0.7812 - binary_accuracy: 0.78 - ETA:
0s - loss: 0.5663 - accuracy: 0.7301 - binary_accuracy: 0.73 - ETA: 0s - loss: 0.5694 - accuracy: 0.7254 - binary_accu
racy: 0.72 - 0s 56us/sample - loss: 0.5643 - accuracy: 0.7376 - binary_accuracy: 0.7376 - val_loss: 0.5491 - val_accur
acy: 0.7748 - val_binary_accuracy: 0.7748
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5424 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.5390 - accuracy: 0.7686 - binary_accuracy: 0.76 - ETA: 0s - loss: 0.5385 - accuracy: 0.7673 - binary_accu
racy: 0.76 - 0s 57us/sample - loss: 0.5384 - accuracy: 0.7699 - binary_accuracy: 0.7699 - val_loss: 0.5243 - val_accur
acy: 0.8038 - val_binary_accuracy: 0.8038
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5244 - accuracy: 0.8125 - binary_accuracy: 0.81 - ETA:
0s - loss: 0.5150 - accuracy: 0.7904 - binary_accuracy: 0.79 - ETA: 0s - loss: 0.5163 - accuracy: 0.7948 - binary_accu
racy: 0.79 - 0s 61us/sample - loss: 0.5142 - accuracy: 0.7950 - binary_accuracy: 0.7950 - val_loss: 0.5007 - val_accur
acy: 0.8154 - val_binary_accuracy: 0.8154
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4907 - accuracy: 0.7812 - binary_accuracy: 0.78 - ETA:
0s - loss: 0.4933 - accuracy: 0.8087 - binary_accuracy: 0.80 - ETA: 0s - loss: 0.4899 - accuracy: 0.8163 - binary_accu
racy: 0.81 - ETA: 0s - loss: 0.4906 - accuracy: 0.8132 - binary_accuracy: 0.81 - 0s 77us/sample - loss: 0.4914 - accur
acy: 0.8130 - binary_accuracy: 0.8130 - val_loss: 0.4789 - val_accuracy: 0.8320 - val_binary_accuracy: 0.8320
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4571 - accuracy: 0.8438 - binary_accuracy: 0.84 - ETA:
0s - loss: 0.4771 - accuracy: 0.8380 - binary_accuracy: 0.83 - ETA: 0s - loss: 0.4747 - accuracy: 0.8297 - binary_accu
racy: 0.82 - ETA: 0s - loss: 0.4735 - accuracy: 0.8273 - binary_accuracy: 0.82 - ETA: 0s - loss: 0.4710 - accuracy: 0.
8333 - binary_accuracy: 0.83 - 0s 129us/sample - loss: 0.4702 - accuracy: 0.8342 - binary_accuracy: 0.8342 - val_loss:
0.4584 - val_accuracy: 0.8414 - val_binary_accuracy: 0.8414
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5210 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.4521 - accuracy: 0.8472 - binary_accuracy: 0.84 - ETA: 0s - loss: 0.4509 - accuracy: 0.8416 - binary_accu
racy: 0.84 - ETA: 0s - loss: 0.4468 - accuracy: 0.8469 - binary_accuracy: 0.84 - ETA: 0s - loss: 0.4506 - accuracy: 0.
```

```
8406 - binary_accuracy: 0.84 - 0s 84us/sample - loss: 0.4507 - accuracy: 0.8404 - binary_accuracy: 0.8404 - val_loss:
0.4399 - val_accuracy: 0.8530 - val_binary_accuracy: 0.8530
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.4704 - accuracy: 0.8750 - binary_accuracy: 0.87 - ETA:
0s - loss: 0.4382 - accuracy: 0.8586 - binary_accuracy: 0.85 - ETA: 0s - loss: 0.4362 - accuracy: 0.8562 - binary_accu
racy: 0.85 - ETA: 0s - loss: 0.4374 - accuracy: 0.8535 - binary_accuracy: 0.85 - 0s 79us/sample - loss: 0.4329 - accur
acy: 0.8556 - binary_accuracy: 0.8556 - val_loss: 0.4227 - val_accuracy: 0.8573 - val_binary_accuracy: 0.8573
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.3628 - accuracy: 0.9062 - binary_accuracy: 0.90 - ETA:
0s - loss: 0.4152 - accuracy: 0.8587 - binary_accuracy: 0.85 - ETA: 0s - loss: 0.4176 - accuracy: 0.8623 - binary_accu
racy: 0.86 - ETA: 0s - loss: 0.4204 - accuracy: 0.8590 - binary_accuracy: 0.85 - 0s 82us/sample - loss: 0.4166 - accur
acy: 0.8621 - binary_accuracy: 0.8621 - val_loss: 0.4072 - val_accuracy: 0.8602 - val_binary_accuracy: 0.8602
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.01

|-units: 30

|-Score: 0.8696596622467041

|-Best step: 0

```
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 48s - loss: 0.7067 - accuracy: 0.4688 - binary_accuracy: 0.468 - ET
A: 0s - loss: 0.6944 - accuracy: 0.5393 - binary_accuracy: 0.539 - ETA: 0s - loss: 0.6926 - accuracy: 0.5537 - binary_
accuracy: 0.55 - 1s 252us/sample - loss: 0.6897 - accuracy: 0.5658 - binary_accuracy: 0.5658 - val_loss: 0.6785 - val_
accuracy: 0.6046 - val_binary_accuracy: 0.6046
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6742 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6824 - accuracy: 0.5958 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6767 - accuracy: 0.6006 - binary_accu
racy: 0.60 - 0s 63us/sample - loss: 0.6726 - accuracy: 0.6012 - binary_accuracy: 0.6012 - val_loss: 0.6663 - val_accur
acy: 0.6025 - val_binary_accuracy: 0.6025
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6249 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.6632 - accuracy: 0.6128 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6627 - accuracy: 0.6114 - binary_accu
racy: 0.61 - 0s 66us/sample - loss: 0.6628 - accuracy: 0.6075 - binary_accuracy: 0.6075 - val_loss: 0.6583 - val_accur
acy: 0.6083 - val_binary_accuracy: 0.6083
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5659 - accuracy: 0.7500 - binary_accuracy: 0.75 - ETA:
0s - loss: 0.6494 - accuracy: 0.6286 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6535 - accuracy: 0.6174 - binary_accu
racy: 0.61 - 0s 63us/sample - loss: 0.6555 - accuracy: 0.6137 - binary_accuracy: 0.6137 - val_loss: 0.6519 - val_accur
acy: 0.6169 - val_binary_accuracy: 0.6169
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7032 - accuracy: 0.4375 - binary_accuracy: 0.43 - ETA:
0s - loss: 0.6465 - accuracy: 0.6293 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6519 - accuracy: 0.6154 - binary_accu
racy: 0.61 - 0s 64us/sample - loss: 0.6492 - accuracy: 0.6193 - binary_accuracy: 0.6193 - val_loss: 0.6459 - val_accur
acy: 0.6213 - val_binary_accuracy: 0.6213
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7038 - accuracy: 0.4688 - binary_accuracy: 0.46 - ETA:
0s - loss: 0.6564 - accuracy: 0.5992 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6462 - accuracy: 0.6208 - binary_accu
racy: 0.62 - 0s 64us/sample - loss: 0.6432 - accuracy: 0.6233 - binary_accuracy: 0.6233 - val_loss: 0.6402 - val_accur
acy: 0.6256 - val_binary_accuracy: 0.6256
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6414 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6389 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6381 - accuracy: 0.6270 - binary_accu
racy: 0.62 - 0s 66us/sample - loss: 0.6374 - accuracy: 0.6292 - binary_accuracy: 0.6292 - val_loss: 0.6347 - val_accur
acy: 0.6264 - val_binary_accuracy: 0.6264
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6602 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6302 - accuracy: 0.6406 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6336 - accuracy: 0.6320 - binary_accu
racy: 0.63 - 0s 65us/sample - loss: 0.6318 - accuracy: 0.6339 - binary_accuracy: 0.6339 - val_loss: 0.6293 - val_accur
acy: 0.6307 - val_binary_accuracy: 0.6307
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6353 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6264 - accuracy: 0.6406 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6232 - accuracy: 0.6462 - binary_accu
racy: 0.64 - ETA: 0s - loss: 0.6245 - accuracy: 0.6412 - binary_accuracy: 0.64 - 0s 101us/sample - loss: 0.6264 - accu
racy: 0.6376 - binary_accuracy: 0.6376 - val_loss: 0.6241 - val_accuracy: 0.6423 - val_binary_accuracy: 0.6423
Epoch 10/10
```

```
3220/3220 [==============================] - ETA: 0s - loss: 0.6406 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6236 - accuracy: 0.6335 - binary_accuracy: 0.63 - ETA: 0s - loss: 0.6242 - accuracy: 0.6348 - binary_accu
racy: 0.63 - 0s 67us/sample - loss: 0.6210 - accuracy: 0.6425 - binary_accuracy: 0.6425 - val_loss: 0.6189 - val_accur
acy: 0.6466 - val_binary_accuracy: 0.6466
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 37s - loss: 0.7521 - accuracy: 0.2812 - binary_accuracy: 0.281 - ET
A: 0s - loss: 0.7248 - accuracy: 0.4535 - binary_accuracy: 0.453 - ETA: 0s - loss: 0.7165 - accuracy: 0.4769 - binary_
accuracy: 0.47 - 1s 208us/sample - loss: 0.7109 - accuracy: 0.4932 - binary_accuracy: 0.4932 - val_loss: 0.6998 - val_
accuracy: 0.5243 - val_binary_accuracy: 0.5243
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6522 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6955 - accuracy: 0.5320 - binary_accuracy: 0.53 - ETA: 0s - loss: 0.6905 - accuracy: 0.5522 - binary_accu
racy: 0.55 - 0s 59us/sample - loss: 0.6875 - accuracy: 0.5584 - binary_accuracy: 0.5584 - val_loss: 0.6837 - val_accur
acy: 0.5670 - val_binary_accuracy: 0.5670
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6458 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6723 - accuracy: 0.5883 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6742 - accuracy: 0.5864 - binary_accu
racy: 0.58 - 0s 57us/sample - loss: 0.6750 - accuracy: 0.5829 - binary_accuracy: 0.5829 - val_loss: 0.6736 - val_accur
acy: 0.5851 - val_binary_accuracy: 0.5851
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6247 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.6728 - accuracy: 0.5841 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6708 - accuracy: 0.5837 - binary_accu
racy: 0.58 - 0s 56us/sample - loss: 0.6663 - accuracy: 0.5929 - binary_accuracy: 0.5929 - val_loss: 0.6657 - val_accur
acy: 0.5894 - val_binary_accuracy: 0.5894
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6193 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6611 - accuracy: 0.6057 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6620 - accuracy: 0.5926 - binary_accu
racy: 0.59 - 0s 55us/sample - loss: 0.6590 - accuracy: 0.5994 - binary_accuracy: 0.5994 - val_loss: 0.6588 - val_accur
acy: 0.5988 - val_binary_accuracy: 0.5988
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6757 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6494 - accuracy: 0.6190 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6523 - accuracy: 0.6052 - binary_accu
racy: 0.60 - 0s 56us/sample - loss: 0.6524 - accuracy: 0.6062 - binary_accuracy: 0.6062 - val_loss: 0.6524 - val_accur
acy: 0.5996 - val_binary_accuracy: 0.5996
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7108 - accuracy: 0.4375 - binary_accuracy: 0.43 - ETA:
0s - loss: 0.6449 - accuracy: 0.6193 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6461 - accuracy: 0.6129 - binary_accu
racy: 0.61 - 0s 54us/sample - loss: 0.6460 - accuracy: 0.6130 - binary_accuracy: 0.6130 - val_loss: 0.6462 - val_accur
acy: 0.6075 - val_binary_accuracy: 0.6075
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6611 - accuracy: 0.5312 - binary_accuracy: 0.53 - ETA:
0s - loss: 0.6395 - accuracy: 0.6265 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6400 - accuracy: 0.6198 - binary_accu
racy: 0.61 - 0s 55us/sample - loss: 0.6399 - accuracy: 0.6171 - binary_accuracy: 0.6171 - val_loss: 0.6402 - val_accur
acy: 0.6177 - val_binary_accuracy: 0.6177
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6671 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6340 - accuracy: 0.6288 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6351 - accuracy: 0.6227 - binary_accu
```

```
racy: 0.62 - 0s 56us/sample - loss: 0.6340 - accuracy: 0.6248 - binary_accuracy: 0.6248 - val_loss: 0.6343 - val_accur
acy: 0.6249 - val_binary_accuracy: 0.6249
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6336 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6324 - accuracy: 0.6186 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6299 - accuracy: 0.6250 - binary_accu
racy: 0.62 - 0s 55us/sample - loss: 0.6282 - accuracy: 0.6295 - binary_accuracy: 0.6295 - val_loss: 0.6286 - val_accur
acy: 0.6314 - val_binary_accuracy: 0.6314
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 37s - loss: 0.6566 - accuracy: 0.5938 - binary_accuracy: 0.593 - ET
A: 0s - loss: 0.7723 - accuracy: 0.4492 - binary_accuracy: 0.449 - ETA: 0s - loss: 0.7541 - accuracy: 0.4497 - binary_
accuracy: 0.44 - 1s 205us/sample - loss: 0.7448 - accuracy: 0.4531 - binary_accuracy: 0.4531 - val_loss: 0.7095 - val_
accuracy: 0.4707 - val_binary_accuracy: 0.4707
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6670 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6994 - accuracy: 0.4789 - binary_accuracy: 0.47 - ETA: 0s - loss: 0.6946 - accuracy: 0.4938 - binary_accu
racy: 0.49 - 0s 65us/sample - loss: 0.6916 - accuracy: 0.5065 - binary_accuracy: 0.5065 - val_loss: 0.6775 - val_accur
acy: 0.5489 - val_binary_accuracy: 0.5489
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6214 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6755 - accuracy: 0.5675 - binary_accuracy: 0.56 - ETA: 0s - loss: 0.6730 - accuracy: 0.5820 - binary_accu
racy: 0.58 - ETA: 0s - loss: 0.6725 - accuracy: 0.5840 - binary_accuracy: 0.58 - 0s 79us/sample - loss: 0.6693 - accur
acy: 0.5932 - binary_accuracy: 0.5932 - val_loss: 0.6620 - val_accuracy: 0.6010 - val_binary_accuracy: 0.6010
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6715 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6590 - accuracy: 0.6230 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6559 - accuracy: 0.6276 - binary_accu
racy: 0.62 - ETA: 0s - loss: 0.6587 - accuracy: 0.6143 - binary_accuracy: 0.61 - 0s 77us/sample - loss: 0.6575 - accur
acy: 0.6177 - binary_accuracy: 0.6177 - val_loss: 0.6531 - val_accuracy: 0.6119 - val_binary_accuracy: 0.6119
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5823 - accuracy: 0.7500 - binary_accuracy: 0.75 - ETA:
0s - loss: 0.6435 - accuracy: 0.6528 - binary_accuracy: 0.65 - ETA: 0s - loss: 0.6516 - accuracy: 0.6376 - binary_accu
racy: 0.63 - ETA: 0s - loss: 0.6508 - accuracy: 0.6348 - binary_accuracy: 0.63 - 0s 81us/sample - loss: 0.6497 - accur
acy: 0.6345 - binary_accuracy: 0.6345 - val_loss: 0.6466 - val_accuracy: 0.6235 - val_binary_accuracy: 0.6235
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6313 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6495 - accuracy: 0.6406 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6479 - accuracy: 0.6339 - binary_accu
racy: 0.63 - ETA: 0s - loss: 0.6434 - accuracy: 0.6401 - binary_accuracy: 0.64 - 0s 77us/sample - loss: 0.6433 - accur
acy: 0.6404 - binary_accuracy: 0.6404 - val_loss: 0.6406 - val_accuracy: 0.6293 - val_binary_accuracy: 0.6293
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5881 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6286 - accuracy: 0.6657 - binary_accuracy: 0.66 - ETA: 0s - loss: 0.6377 - accuracy: 0.6416 - binary_accu
racy: 0.64 - ETA: 0s - loss: 0.6376 - accuracy: 0.6379 - binary_accuracy: 0.63 - 0s 71us/sample - loss: 0.6374 - accur
acy: 0.6398 - binary_accuracy: 0.6398 - val_loss: 0.6351 - val_accuracy: 0.6358 - val_binary_accuracy: 0.6358
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6500 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6242 - accuracy: 0.6736 - binary_accuracy: 0.67 - ETA: 0s - loss: 0.6311 - accuracy: 0.6465 - binary_accu
racy: 0.64 - ETA: 0s - loss: 0.6310 - accuracy: 0.6467 - binary_accuracy: 0.64 - 0s 72us/sample - loss: 0.6317 - accur
acy: 0.6457 - binary_accuracy: 0.6457 - val_loss: 0.6297 - val_accuracy: 0.6430 - val_binary_accuracy: 0.6430
```

```
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6511 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6325 - accuracy: 0.6464 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6294 - accuracy: 0.6467 - binary_accu
racy: 0.64 - ETA: 0s - loss: 0.6255 - accuracy: 0.6525 - binary_accuracy: 0.65 - 0s 79us/sample - loss: 0.6263 - accur
acy: 0.6481 - binary_accuracy: 0.6481 - val_loss: 0.6245 - val_accuracy: 0.6488 - val_binary_accuracy: 0.6488
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5866 - accuracy: 0.7812 - binary_accuracy: 0.78 - ETA:
0s - loss: 0.6230 - accuracy: 0.6447 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6232 - accuracy: 0.6538 - binary_accu
racy: 0.65 - 0s 64us/sample - loss: 0.6209 - accuracy: 0.6562 - binary_accuracy: 0.6562 - val_loss: 0.6194 - val_accur
acy: 0.6568 - val_binary_accuracy: 0.6568
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.001

|-units: 82

|-Score: 0.6449432969093323

|-Best step: 0

```
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 39s - loss: 0.6979 - accuracy: 0.5625 - binary_accuracy: 0.562 - ET
A: 0s - loss: 0.6740 - accuracy: 0.5785 - binary_accuracy: 0.578 - ETA: 0s - loss: 0.6711 - accuracy: 0.5962 - binary_
accuracy: 0.59 - 1s 219us/sample - loss: 0.6691 - accuracy: 0.6065 - binary_accuracy: 0.6065 - val_loss: 0.6622 - val_
accuracy: 0.6097 - val_binary_accuracy: 0.6097
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6398 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6590 - accuracy: 0.6380 - binary_accuracy: 0.63 - ETA: 0s - loss: 0.6557 - accuracy: 0.6461 - binary_accu
racy: 0.64 - 0s 67us/sample - loss: 0.6543 - accuracy: 0.6478 - binary_accuracy: 0.6478 - val_loss: 0.6513 - val_accur
acy: 0.6488 - val_binary_accuracy: 0.6488
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6749 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6504 - accuracy: 0.6484 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6467 - accuracy: 0.6483 - binary_accu
racy: 0.64 - 0s 67us/sample - loss: 0.6458 - accuracy: 0.6500 - binary_accuracy: 0.6500 - val_loss: 0.6441 - val_accur
acy: 0.6575 - val_binary_accuracy: 0.6575
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7146 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 0.6411 - accuracy: 0.6445 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6422 - accuracy: 0.6477 - binary_accu
racy: 0.64 - 0s 65us/sample - loss: 0.6395 - accuracy: 0.6506 - binary_accuracy: 0.6506 - val_loss: 0.6381 - val_accur
acy: 0.6604 - val_binary_accuracy: 0.6604
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6351 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6392 - accuracy: 0.6417 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6342 - accuracy: 0.6511 - binary_accu
racy: 0.65 - 0s 64us/sample - loss: 0.6340 - accuracy: 0.6522 - binary_accuracy: 0.6522 - val_loss: 0.6327 - val_accur
acy: 0.6676 - val_binary_accuracy: 0.6676
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6146 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6319 - accuracy: 0.6532 - binary_accuracy: 0.65 - ETA: 0s - loss: 0.6282 - accuracy: 0.6562 - binary_accu
racy: 0.65 - 0s 64us/sample - loss: 0.6288 - accuracy: 0.6547 - binary_accuracy: 0.6547 - val_loss: 0.6276 - val_accur
acy: 0.6734 - val_binary_accuracy: 0.6734
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6420 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6247 - accuracy: 0.6600 - binary_accuracy: 0.66 - ETA: 0s - loss: 0.6255 - accuracy: 0.6570 - binary_accu
racy: 0.65 - 0s 65us/sample - loss: 0.6237 - accuracy: 0.6596 - binary_accuracy: 0.6596 - val_loss: 0.6226 - val_accur
acy: 0.6785 - val_binary_accuracy: 0.6785
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6181 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6026 - accuracy: 0.6916 - binary_accuracy: 0.69 - ETA: 0s - loss: 0.6124 - accuracy: 0.6713 - binary_accu
racy: 0.67 - 0s 67us/sample - loss: 0.6187 - accuracy: 0.6640 - binary_accuracy: 0.6640 - val_loss: 0.6178 - val_accur
acy: 0.6807 - val_binary_accuracy: 0.6807
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6642 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6152 - accuracy: 0.6726 - binary_accuracy: 0.67 - ETA: 0s - loss: 0.6155 - accuracy: 0.6675 - binary_accu
racy: 0.66 - 0s 66us/sample - loss: 0.6139 - accuracy: 0.6711 - binary_accuracy: 0.6711 - val_loss: 0.6130 - val_accur
acy: 0.6857 - val_binary_accuracy: 0.6857
Epoch 10/10
```

```
3220/3220 [==============================] - ETA: 0s - loss: 0.5701 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA: 
0s - loss: 0.6220 - accuracy: 0.6639 - binary_accuracy: 0.66 - ETA: 0s - loss: 0.6110 - accuracy: 0.6788 - binary_accu
racy: 0.67 - 0s 70us/sample - loss: 0.6091 - accuracy: 0.6811 - binary_accuracy: 0.6811 - val_loss: 0.6082 - val_accur
acy: 0.6894 - val_binary_accuracy: 0.6894
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 36s - loss: 0.6163 - accuracy: 0.7500 - binary_accuracy: 0.750 - ET
A: 0s - loss: 0.6565 - accuracy: 0.6601 - binary_accuracy: 0.660 - ETA: 0s - loss: 0.6560 - accuracy: 0.6547 - binary_
accuracy: 0.65 - 1s 200us/sample - loss: 0.6557 - accuracy: 0.6531 - binary_accuracy: 0.6531 - val_loss: 0.6464 - val_
accuracy: 0.6626 - val_binary_accuracy: 0.6626
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6451 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA: 
0s - loss: 0.6496 - accuracy: 0.6598 - binary_accuracy: 0.65 - ETA: 0s - loss: 0.6458 - accuracy: 0.6668 - binary_accu
racy: 0.66 - 0s 59us/sample - loss: 0.6483 - accuracy: 0.6590 - binary_accuracy: 0.6590 - val_loss: 0.6397 - val_accur
acy: 0.6749 - val_binary_accuracy: 0.6749
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5788 - accuracy: 0.8125 - binary_accuracy: 0.81 - ETA: 
0s - loss: 0.6435 - accuracy: 0.6689 - binary_accuracy: 0.66 - ETA: 0s - loss: 0.6411 - accuracy: 0.6685 - binary_accu
racy: 0.66 - 0s 55us/sample - loss: 0.6420 - accuracy: 0.6630 - binary_accuracy: 0.6630 - val_loss: 0.6338 - val_accur
acy: 0.6734 - val_binary_accuracy: 0.6734
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6855 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.6418 - accuracy: 0.6584 - binary_accuracy: 0.65 - ETA: 0s - loss: 0.6380 - accuracy: 0.6684 - binary_accu
racy: 0.66 - 0s 55us/sample - loss: 0.6361 - accuracy: 0.6668 - binary_accuracy: 0.6668 - val_loss: 0.6282 - val_accur
acy: 0.6698 - val_binary_accuracy: 0.6698
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6305 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA: 
0s - loss: 0.6434 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA: 0s - loss: 0.6342 - accuracy: 0.6635 - binary_accu
racy: 0.66 - 0s 54us/sample - loss: 0.6304 - accuracy: 0.6711 - binary_accuracy: 0.6711 - val_loss: 0.6228 - val_accur
acy: 0.6720 - val_binary_accuracy: 0.6720
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6347 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA: 
0s - loss: 0.6298 - accuracy: 0.6690 - binary_accuracy: 0.66 - ETA: 0s - loss: 0.6262 - accuracy: 0.6732 - binary_accu
racy: 0.67 - 0s 55us/sample - loss: 0.6249 - accuracy: 0.6727 - binary_accuracy: 0.6727 - val_loss: 0.6175 - val_accur
acy: 0.6799 - val_binary_accuracy: 0.6799
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5858 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA: 
0s - loss: 0.6257 - accuracy: 0.6737 - binary_accuracy: 0.67 - ETA: 0s - loss: 0.6219 - accuracy: 0.6795 - binary_accu
racy: 0.67 - 0s 57us/sample - loss: 0.6196 - accuracy: 0.6817 - binary_accuracy: 0.6817 - val_loss: 0.6124 - val_accur
acy: 0.6857 - val_binary_accuracy: 0.6857
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5952 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA: 
0s - loss: 0.6117 - accuracy: 0.6969 - binary_accuracy: 0.69 - ETA: 0s - loss: 0.6137 - accuracy: 0.6835 - binary_accu
racy: 0.68 - 0s 62us/sample - loss: 0.6144 - accuracy: 0.6842 - binary_accuracy: 0.6842 - val_loss: 0.6073 - val_accur
acy: 0.6937 - val_binary_accuracy: 0.6937
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6385 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.6169 - accuracy: 0.6845 - binary_accuracy: 0.68 - ETA: 0s - loss: 0.6066 - accuracy: 0.7001 - binary_accu
```

```
racy: 0.70 - 0s 65us/sample - loss: 0.6092 - accuracy: 0.6932 - binary_accuracy: 0.6932 - val_loss: 0.6024 - val_accur
acy: 0.7031 - val_binary_accuracy: 0.7031
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5964 - accuracy: 0.8438 - binary_accuracy: 0.84 - ETA:
0s - loss: 0.5979 - accuracy: 0.7228 - binary_accuracy: 0.72 - ETA: 0s - loss: 0.5991 - accuracy: 0.7079 - binary_accu
racy: 0.70 - 0s 69us/sample - loss: 0.6042 - accuracy: 0.6984 - binary_accuracy: 0.6984 - val_loss: 0.5975 - val_accur
acy: 0.7082 - val_binary_accuracy: 0.7082
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 37s - loss: 0.7037 - accuracy: 0.5625 - binary_accuracy: 0.562 - ET
A: 0s - loss: 0.6951 - accuracy: 0.5961 - binary_accuracy: 0.596 - ETA: 0s - loss: 0.6928 - accuracy: 0.5869 - binary_
accuracy: 0.58 - 1s 204us/sample - loss: 0.6876 - accuracy: 0.5885 - binary_accuracy: 0.5885 - val_loss: 0.6758 - val_
accuracy: 0.5844 - val_binary_accuracy: 0.5844
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7330 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 0.6705 - accuracy: 0.5945 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6705 - accuracy: 0.5849 - binary_accu
racy: 0.58 - 0s 58us/sample - loss: 0.6728 - accuracy: 0.5786 - binary_accuracy: 0.5786 - val_loss: 0.6646 - val_accur
acy: 0.5858 - val_binary_accuracy: 0.5858
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6575 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6624 - accuracy: 0.5861 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6624 - accuracy: 0.5831 - binary_accu
racy: 0.58 - 0s 60us/sample - loss: 0.6634 - accuracy: 0.5773 - binary_accuracy: 0.5773 - val_loss: 0.6566 - val_accur
acy: 0.5844 - val_binary_accuracy: 0.5844
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6892 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 0.6539 - accuracy: 0.5872 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6569 - accuracy: 0.5806 - binary_accu
racy: 0.58 - 0s 59us/sample - loss: 0.6560 - accuracy: 0.5770 - binary_accuracy: 0.5770 - val_loss: 0.6499 - val_accur
acy: 0.5865 - val_binary_accuracy: 0.5865
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6373 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6493 - accuracy: 0.5828 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6545 - accuracy: 0.5724 - binary_accu
racy: 0.57 - 0s 58us/sample - loss: 0.6493 - accuracy: 0.5823 - binary_accuracy: 0.5823 - val_loss: 0.6437 - val_accur
acy: 0.5873 - val_binary_accuracy: 0.5873
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6012 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6461 - accuracy: 0.5861 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6431 - accuracy: 0.5875 - binary_accu
racy: 0.58 - 0s 59us/sample - loss: 0.6431 - accuracy: 0.5866 - binary_accuracy: 0.5866 - val_loss: 0.6378 - val_accur
acy: 0.5923 - val_binary_accuracy: 0.5923
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6436 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6342 - accuracy: 0.6067 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6351 - accuracy: 0.6019 - binary_accu
racy: 0.60 - 0s 57us/sample - loss: 0.6371 - accuracy: 0.5957 - binary_accuracy: 0.5957 - val_loss: 0.6320 - val_accur
acy: 0.6025 - val_binary_accuracy: 0.6025
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5935 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6359 - accuracy: 0.5977 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6340 - accuracy: 0.5977 - binary_accu
racy: 0.59 - 0s 57us/sample - loss: 0.6311 - accuracy: 0.6053 - binary_accuracy: 0.6053 - val_loss: 0.6263 - val_accur
acy: 0.6133 - val_binary_accuracy: 0.6133
```

```
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6486 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.6236 - accuracy: 0.6273 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6267 - accuracy: 0.6100 - binary_accu
racy: 0.61 - 0s 58us/sample - loss: 0.6254 - accuracy: 0.6121 - binary_accuracy: 0.6121 - val_loss: 0.6208 - val_accur
acy: 0.6271 - val_binary_accuracy: 0.6271
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6472 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.6207 - accuracy: 0.6124 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6211 - accuracy: 0.6139 - binary_accu
racy: 0.61 - 0s 56us/sample - loss: 0.6198 - accuracy: 0.6189 - binary_accuracy: 0.6189 - val_loss: 0.6154 - val_accur
acy: 0.6307 - val_binary_accuracy: 0.6307
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.001

|-units: 88

|-Score: 0.676080048084259

|-Best step: 0

```
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 37s - loss: 0.7316 - accuracy: 0.4688 - binary_accuracy: 0.468 - ET
A: 0s - loss: 0.7599 - accuracy: 0.4199 - binary_accuracy: 0.419 - ETA: 0s - loss: 0.7643 - accuracy: 0.3947 - binary_
accuracy: 0.39 - 1s 216us/sample - loss: 0.7630 - accuracy: 0.3938 - binary_accuracy: 0.3938 - val_loss: 0.7671 - val_
accuracy: 0.3845 - val_binary_accuracy: 0.3845
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7992 - accuracy: 0.4375 - binary_accuracy: 0.43 - ETA:
0s - loss: 0.7582 - accuracy: 0.3971 - binary_accuracy: 0.39 - ETA: 0s - loss: 0.7576 - accuracy: 0.3898 - binary_accu
racy: 0.38 - 0s 60us/sample - loss: 0.7547 - accuracy: 0.3953 - binary_accuracy: 0.3953 - val_loss: 0.7590 - val_accur
acy: 0.3838 - val_binary_accuracy: 0.3838
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.8158 - accuracy: 0.2188 - binary_accuracy: 0.21 - ETA:
0s - loss: 0.7446 - accuracy: 0.4121 - binary_accuracy: 0.41 - ETA: 0s - loss: 0.7477 - accuracy: 0.4032 - binary_accu
racy: 0.40 - 0s 65us/sample - loss: 0.7471 - accuracy: 0.4040 - binary_accuracy: 0.4040 - val_loss: 0.7516 - val_accur
acy: 0.3874 - val_binary_accuracy: 0.3874
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7631 - accuracy: 0.2812 - binary_accuracy: 0.28 - ETA:
0s - loss: 0.7417 - accuracy: 0.4055 - binary_accuracy: 0.40 - ETA: 0s - loss: 0.7418 - accuracy: 0.4030 - binary_accu
racy: 0.40 - 0s 65us/sample - loss: 0.7402 - accuracy: 0.4053 - binary_accuracy: 0.4053 - val_loss: 0.7449 - val_accur
acy: 0.3925 - val_binary_accuracy: 0.3925
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7386 - accuracy: 0.3750 - binary_accuracy: 0.37 - ETA:
0s - loss: 0.7334 - accuracy: 0.4040 - binary_accuracy: 0.40 - ETA: 0s - loss: 0.7320 - accuracy: 0.4093 - binary_accu
racy: 0.40 - 0s 65us/sample - loss: 0.7339 - accuracy: 0.4087 - binary_accuracy: 0.4087 - val_loss: 0.7388 - val_accur
acy: 0.3968 - val_binary_accuracy: 0.3968
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6628 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA:
0s - loss: 0.7210 - accuracy: 0.4360 - binary_accuracy: 0.43 - ETA: 0s - loss: 0.7268 - accuracy: 0.4224 - binary_accu
racy: 0.42 - 0s 65us/sample - loss: 0.7282 - accuracy: 0.4199 - binary_accuracy: 0.4199 - val_loss: 0.7332 - val_accur
acy: 0.4120 - val_binary_accuracy: 0.4120
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7396 - accuracy: 0.3750 - binary_accuracy: 0.37 - ETA:
0s - loss: 0.7286 - accuracy: 0.4238 - binary_accuracy: 0.42 - ETA: 0s - loss: 0.7240 - accuracy: 0.4225 - binary_accu
racy: 0.42 - 0s 67us/sample - loss: 0.7229 - accuracy: 0.4248 - binary_accuracy: 0.4248 - val_loss: 0.7281 - val_accur
acy: 0.4164 - val_binary_accuracy: 0.4164
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6960 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 0.7180 - accuracy: 0.4286 - binary_accuracy: 0.42 - ETA: 0s - loss: 0.7204 - accuracy: 0.4328 - binary_accu
racy: 0.43 - 0s 67us/sample - loss: 0.7182 - accuracy: 0.4295 - binary_accuracy: 0.4295 - val_loss: 0.7235 - val_accur
acy: 0.4258 - val_binary_accuracy: 0.4258
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7186 - accuracy: 0.4062 - binary_accuracy: 0.40 - ETA:
0s - loss: 0.7154 - accuracy: 0.4505 - binary_accuracy: 0.45 - ETA: 0s - loss: 0.7134 - accuracy: 0.4539 - binary_accu
racy: 0.45 - 0s 67us/sample - loss: 0.7138 - accuracy: 0.4503 - binary_accuracy: 0.4503 - val_loss: 0.7192 - val_accur
acy: 0.4424 - val_binary_accuracy: 0.4424
Epoch 10/10
```

```
3220/3220 [==============================] - ETA: 0s - loss: 0.6794 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA:
0s - loss: 0.7076 - accuracy: 0.4531 - binary_accuracy: 0.45 - ETA: 0s - loss: 0.7094 - accuracy: 0.4639 - binary_accu
racy: 0.46 - 0s 67us/sample - loss: 0.7099 - accuracy: 0.4686 - binary_accuracy: 0.4686 - val_loss: 0.7153 - val_accur
acy: 0.4736 - val_binary_accuracy: 0.4736
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 54s - loss: 0.4705 - accuracy: 0.8438 - binary_accuracy: 0.843 - ET
A: 0s - loss: 0.6655 - accuracy: 0.6098 - binary_accuracy: 0.609 - ETA: 0s - loss: 0.6689 - accuracy: 0.6082 - binary_
accuracy: 0.60 - 1s 273us/sample - loss: 0.6696 - accuracy: 0.6071 - binary_accuracy: 0.6071 - val_loss: 0.6675 - val_
accuracy: 0.6090 - val_binary_accuracy: 0.6090
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5715 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6668 - accuracy: 0.6106 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6744 - accuracy: 0.6011 - binary_accu
racy: 0.60 - ETA: 0s - loss: 0.6629 - accuracy: 0.6132 - binary_accuracy: 0.61 - 0s 93us/sample - loss: 0.6668 - accur
acy: 0.6096 - binary_accuracy: 0.6096 - val_loss: 0.6649 - val_accuracy: 0.6104 - val_binary_accuracy: 0.6104
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6912 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6565 - accuracy: 0.6159 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6646 - accuracy: 0.6107 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6640 - accuracy: 0.6097 - binary_accuracy: 0.60 - 0s 75us/sample - loss: 0.6643 - accur
acy: 0.6102 - binary_accuracy: 0.6102 - val_loss: 0.6625 - val_accuracy: 0.6119 - val_binary_accuracy: 0.6119
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.5812 - accuracy: 0.7188 - binary_accuracy: 0.71 - ETA:
0s - loss: 0.6545 - accuracy: 0.6166 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6638 - accuracy: 0.6127 - binary_accu
racy: 0.61 - 0s 65us/sample - loss: 0.6619 - accuracy: 0.6134 - binary_accuracy: 0.6134 - val_loss: 0.6602 - val_accur
acy: 0.6162 - val_binary_accuracy: 0.6162
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6132 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA:
0s - loss: 0.6677 - accuracy: 0.6066 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6623 - accuracy: 0.6167 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6597 - accuracy: 0.6171 - binary_accuracy: 0.61 - 0s 81us/sample - loss: 0.6596 - accur
acy: 0.6168 - binary_accuracy: 0.6168 - val_loss: 0.6581 - val_accuracy: 0.6191 - val_binary_accuracy: 0.6191
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6317 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA:
0s - loss: 0.6616 - accuracy: 0.6149 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6590 - accuracy: 0.6155 - binary_accu
racy: 0.61 - 0s 67us/sample - loss: 0.6575 - accuracy: 0.6174 - binary_accuracy: 0.6174 - val_loss: 0.6561 - val_accur
acy: 0.6191 - val_binary_accuracy: 0.6191
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6974 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA:
0s - loss: 0.6681 - accuracy: 0.6078 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6717 - accuracy: 0.5984 - binary_accu
racy: 0.59 - ETA: 0s - loss: 0.6654 - accuracy: 0.6049 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6623 - accuracy: 0.
6091 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6558 - accuracy: 0.6173 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6
546 - accuracy: 0.6202 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6519 - accuracy: 0.6240 - binary_accuracy: 0.62 - 1
s 203us/sample - loss: 0.6555 - accuracy: 0.6183 - binary_accuracy: 0.6183 - val_loss: 0.6542 - val_accuracy: 0.6235 -
val_binary_accuracy: 0.6235
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.7335 - accuracy: 0.5000 - binary_accuracy: 0.50 - ETA:
0s - loss: 0.6637 - accuracy: 0.6000 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6704 - accuracy: 0.5974 - binary_accu
racy: 0.59 - ETA: 0s - loss: 0.6595 - accuracy: 0.6125 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6587 - accuracy: 0.
6118 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6542 - accuracy: 0.6174 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6
```

```
516 - accuracy: 0.6246 - binary_accuracy: 0.62 - 1s 161us/sample - loss: 0.6536 - accuracy: 0.6214 - binary_accuracy: 
0.6214 - val_loss: 0.6525 - val_accuracy: 0.6242 - val_binary_accuracy: 0.6242
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6102 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA: 
0s - loss: 0.6641 - accuracy: 0.6060 - binary_accuracy: 0.60 - ETA: 0s - loss: 0.6551 - accuracy: 0.6171 - binary_accu
racy: 0.61 - ETA: 0s - loss: 0.6534 - accuracy: 0.6204 - binary_accuracy: 0.62 - 0s 89us/sample - loss: 0.6519 - accur
acy: 0.6239 - binary_accuracy: 0.6239 - val_loss: 0.6508 - val_accuracy: 0.6249 - val_binary_accuracy: 0.6249
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6507 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.6353 - accuracy: 0.6463 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6403 - accuracy: 0.6431 - binary_accu
racy: 0.64 - ETA: 0s - loss: 0.6490 - accuracy: 0.6254 - binary_accuracy: 0.62 - 0s 83us/sample - loss: 0.6502 - accur
acy: 0.6252 - binary_accuracy: 0.6252 - val_loss: 0.6492 - val_accuracy: 0.6249 - val_binary_accuracy: 0.6249
Train on 3220 samples, validate on 1381 samples
Epoch 1/10
3220/3220 [==============================] - ETA: 42s - loss: 0.6774 - accuracy: 0.5312 - binary_accuracy: 0.531 - ET
A: 0s - loss: 0.6731 - accuracy: 0.5570 - binary_accuracy: 0.557 - ETA: 0s - loss: 0.6751 - accuracy: 0.5642 - binary_
accuracy: 0.56 - 1s 222us/sample - loss: 0.6724 - accuracy: 0.5717 - binary_accuracy: 0.5717 - val_loss: 0.6668 - val_
accuracy: 0.5764 - val_binary_accuracy: 0.5764
Epoch 2/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6407 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA: 
0s - loss: 0.6636 - accuracy: 0.5884 - binary_accuracy: 0.58 - ETA: 0s - loss: 0.6633 - accuracy: 0.5933 - binary_accu
racy: 0.59 - 0s 60us/sample - loss: 0.6664 - accuracy: 0.5901 - binary_accuracy: 0.5901 - val_loss: 0.6611 - val_accur
acy: 0.6032 - val_binary_accuracy: 0.6032
Epoch 3/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6817 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.6614 - accuracy: 0.5994 - binary_accuracy: 0.59 - ETA: 0s - loss: 0.6628 - accuracy: 0.6025 - binary_accu
racy: 0.60 - 0s 57us/sample - loss: 0.6610 - accuracy: 0.6109 - binary_accuracy: 0.6109 - val_loss: 0.6558 - val_accur
acy: 0.6227 - val_binary_accuracy: 0.6227
Epoch 4/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6237 - accuracy: 0.6562 - binary_accuracy: 0.65 - ETA: 
0s - loss: 0.6557 - accuracy: 0.6198 - binary_accuracy: 0.61 - ETA: 0s - loss: 0.6556 - accuracy: 0.6299 - binary_accu
racy: 0.62 - 0s 65us/sample - loss: 0.6560 - accuracy: 0.6252 - binary_accuracy: 0.6252 - val_loss: 0.6511 - val_accur
acy: 0.6437 - val_binary_accuracy: 0.6437
Epoch 5/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6350 - accuracy: 0.5938 - binary_accuracy: 0.59 - ETA: 
0s - loss: 0.6475 - accuracy: 0.6471 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6520 - accuracy: 0.6431 - binary_accu
racy: 0.64 - 0s 67us/sample - loss: 0.6516 - accuracy: 0.6441 - binary_accuracy: 0.6441 - val_loss: 0.6468 - val_accur
acy: 0.6698 - val_binary_accuracy: 0.6698
Epoch 6/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6065 - accuracy: 0.7812 - binary_accuracy: 0.78 - ETA: 
0s - loss: 0.6529 - accuracy: 0.6482 - binary_accuracy: 0.64 - ETA: 0s - loss: 0.6491 - accuracy: 0.6567 - binary_accu
racy: 0.65 - 0s 73us/sample - loss: 0.6475 - accuracy: 0.6590 - binary_accuracy: 0.6590 - val_loss: 0.6429 - val_accur
acy: 0.6894 - val_binary_accuracy: 0.6894
Epoch 7/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6864 - accuracy: 0.5625 - binary_accuracy: 0.56 - ETA: 
0s - loss: 0.6449 - accuracy: 0.6757 - binary_accuracy: 0.67 - ETA: 0s - loss: 0.6466 - accuracy: 0.6757 - binary_accu
racy: 0.67 - 0s 69us/sample - loss: 0.6438 - accuracy: 0.6801 - binary_accuracy: 0.6801 - val_loss: 0.6393 - val_accur
acy: 0.6980 - val_binary_accuracy: 0.6980
```

```
Epoch 8/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6454 - accuracy: 0.6875 - binary_accuracy: 0.68 - ETA: 0s - loss: 0.6432 - accuracy: 0.6815 - binary_accuracy: 0.68 - ETA: 0s - loss: 0.6427 - accuracy: 0.6871 - binary_accuracy: 0.68 - 0s 68us/sample - loss: 0.6404 - accuracy: 0.6910 - binary_accuracy: 0.6910 - val_loss: 0.6361 - val_accuracy: 0.7161 - val_binary_accuracy: 0.7161
Epoch 9/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6533 - accuracy: 0.6250 - binary_accuracy: 0.62 - ETA: 0s - loss: 0.6364 - accuracy: 0.7214 - binary_accuracy: 0.72 - ETA: 0s - loss: 0.6384 - accuracy: 0.7146 - binary_accuracy: 0.71 - 0s 68us/sample - loss: 0.6373 - accuracy: 0.7134 - binary_accuracy: 0.7134 - val_loss: 0.6331 - val_accuracy: 0.7335 - val_binary_accuracy: 0.7335
Epoch 10/10
3220/3220 [==============================] - ETA: 0s - loss: 0.6659 - accuracy: 0.7500 - binary_accuracy: 0.75 - ETA: 0s - loss: 0.6377 - accuracy: 0.7257 - binary_accuracy: 0.72 - ETA: 0s - loss: 0.6318 - accuracy: 0.7332 - binary_accuracy: 0.73 - ETA: 0s - loss: 0.6345 - accuracy: 0.7207 - binary_accuracy: 0.72 - ETA: 0s - loss: 0.6349 - accuracy: 0.7300 - binary_accuracy: 0.73 - 0s 110us/sample - loss: 0.6345 - accuracy: 0.7311 - binary_accuracy: 0.7311 - val_loss: 0.6304 - val_accuracy: 0.7567 - val_binary_accuracy: 0.7567
```

Trial complete

Trial summary

Hp values:

|-learning_rate: 0.0001

|-units: 84

|-Score: 0.6183924674987793

|-Best step: 0

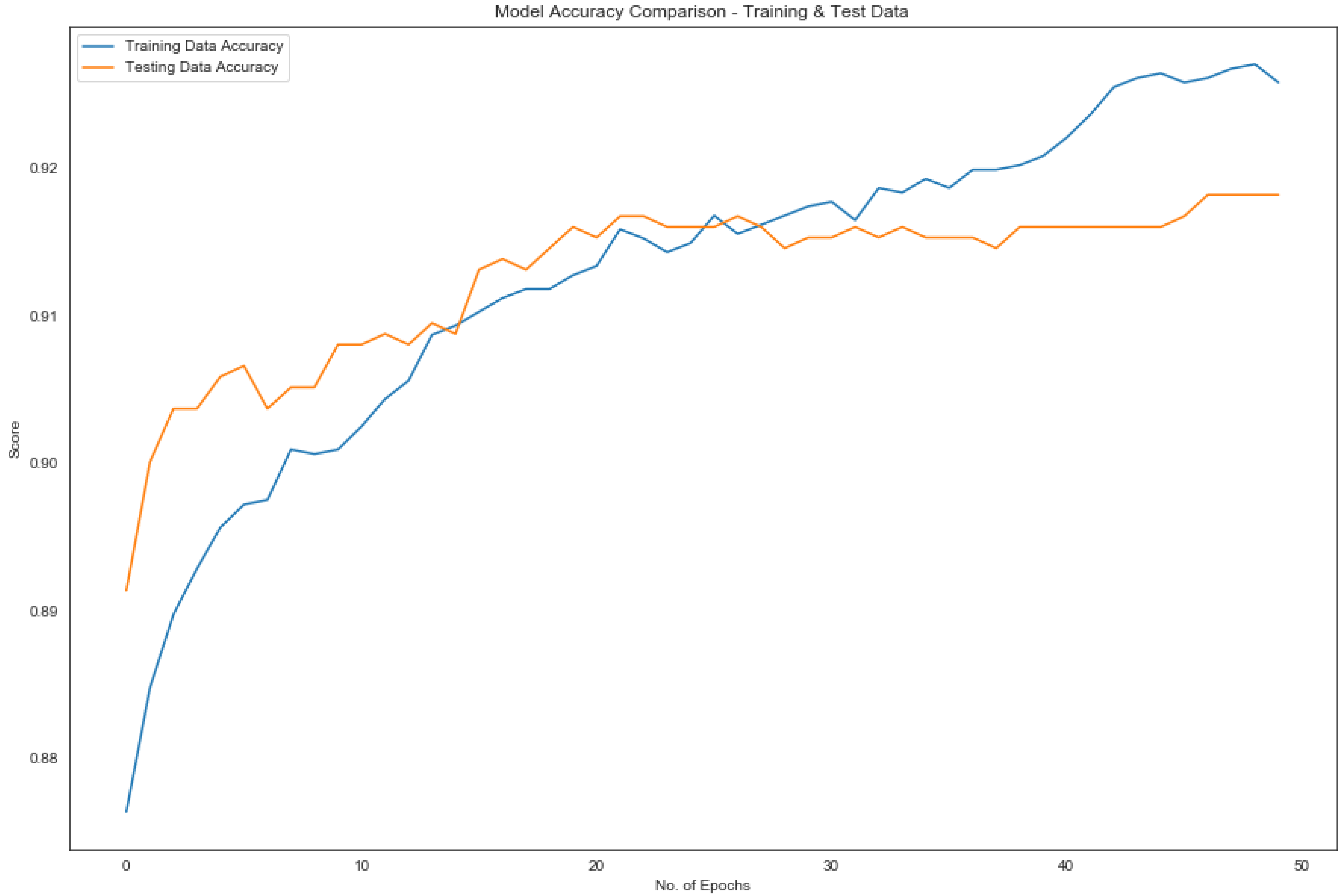
In [123]:
```
best_linear_model = classtuner.get_best_models(num_models = 1)

best_linear_model = best_linear_model[0]
```

In [124]:
```
### Use our best model and reverify it against our test data

spam_linear_history = best_linear_model.fit(X_train_standard, y_train,
                validation_data = (X_test_standard, y_test), epochs = 50, verbose = 0)
```

In [125]:
```
train_acc = best_linear_model.evaluate(X_train_standard, y_train, verbose=0)
test_acc = best_linear_model.evaluate(X_test_standard, y_test, verbose=0)
```

In [126]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Plot scores on each trial for nested CV

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot nested scores for each classifier - quickly visual the best performing model
## This is WITHOUT having changed any of the default parameters
plt.plot(spam_linear_history.history['accuracy'], label = "Training Data Accuracy")
plt.plot(spam_linear_history.history['val_accuracy'], label = "Testing Data Accuracy")

## Give some labels and title
plt.xlabel("No. of Epochs")
plt.ylabel("Score")

## Title and Legend
plt.title("Model Accuracy Comparison - Training & Test Data")
plt.legend()

## Show the graph
plt.show()
```

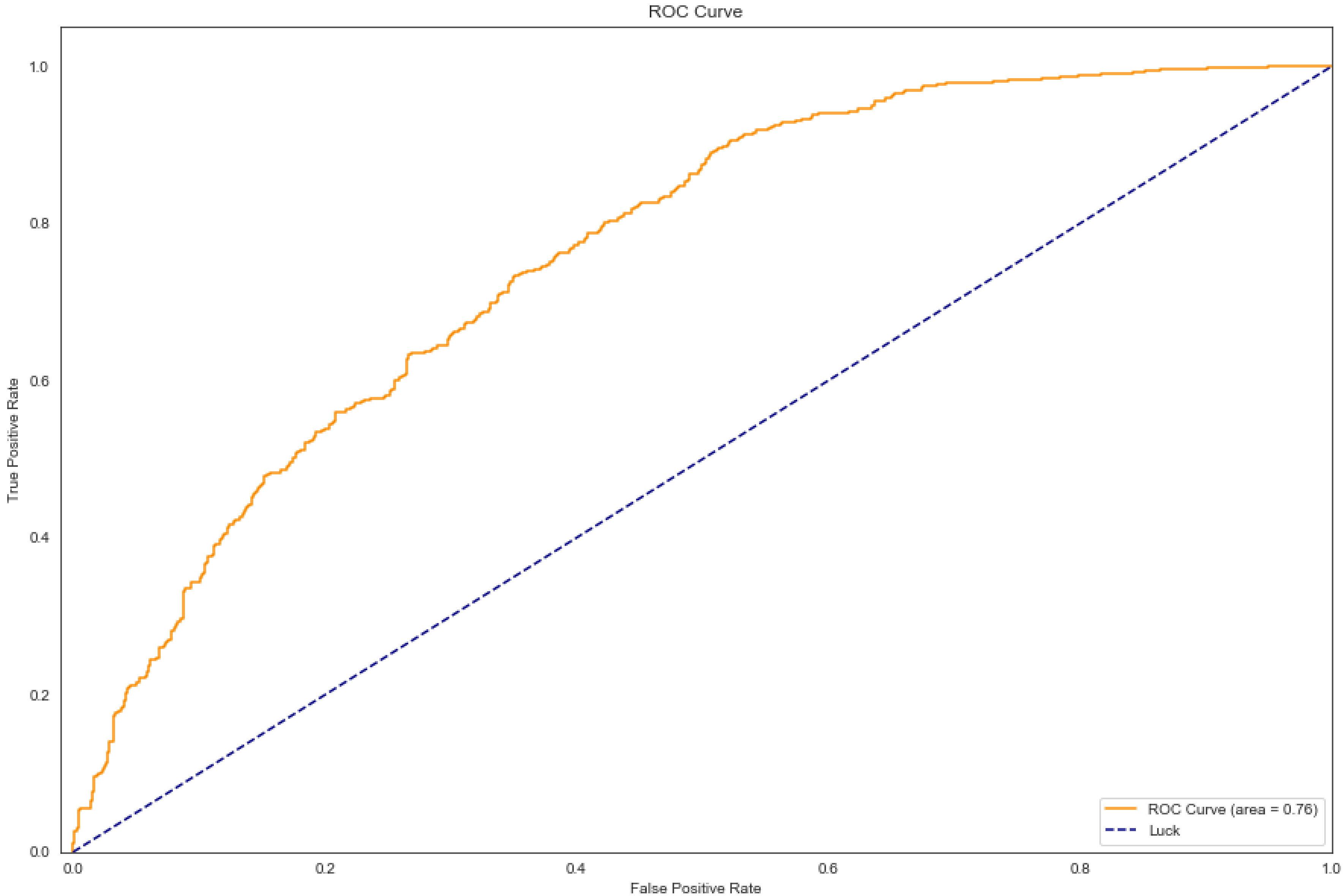


In [127]:
```
## Create three empty lists to store my precision, recall, and average precision scores
precision, recall, average_precision = [], [], []

## Get probabilities for our labels
probas_ = best_linear_model.predict_proba(X_test)
```

In [128]:
```
## Create my false positive rate, true positive rate, and threshold using my test data
fpr, tpr, threholds = roc_curve(y_test, linear_probas_, pos_label = 1)

## Create precision and recall scores to plot with
precision_score, recall_score, _ = precision_recall_curve(y_test, probas_)

## Calculate the overall AUC for the model
auc = np.trapz(tpr, fpr)

## Save the average precision for our model
avg_precision = average_precision_score(y_test, probas_)
```

In [129]:

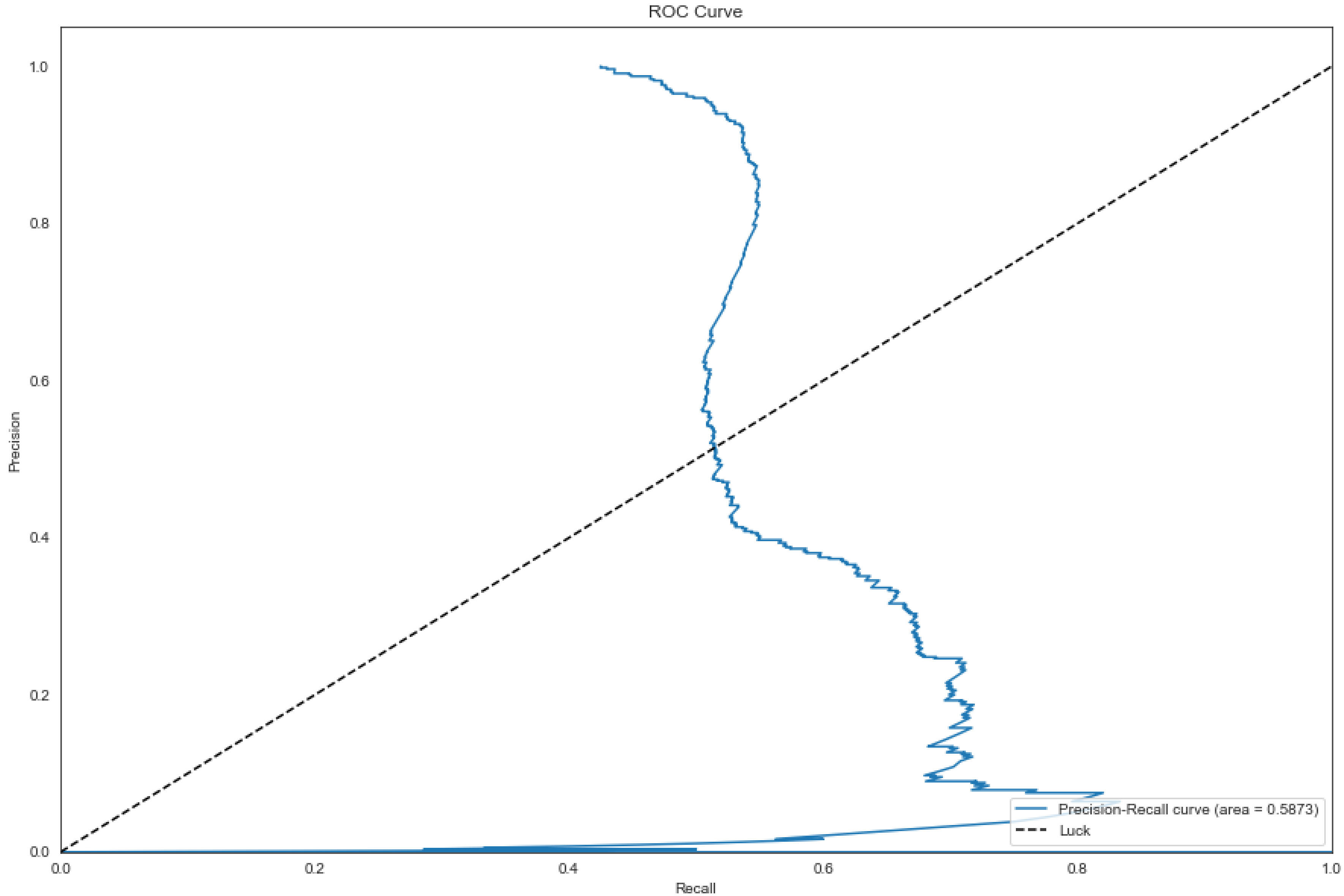
```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Create a new figure to plot
plt.figure(figsize= (15, 10))

lw = 2

## Draw the line for my fpr and tpr
plt.plot(fpr, tpr, color = 'darkorange',
        label = 'ROC Curve (area = %0.2f)' % auc)

## Put in a line to demonstrate blind luck
plt.plot([0, 1], [0, 1], color = 'navy', linestyle = '--', label = 'Luck')

## Set the limits of the plot for better visualization
plt.xlim([-0.01, 1.0])
plt.ylim([0.0, 1.05])

## Set labels for x and y
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')

## Set a title and legend
plt.title('ROC Curve')
plt.legend(loc = 'lower right')

## Show the curve!
plt.show()
```



ROC Curve
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
ROC Curve (area = 0.76)
Luck

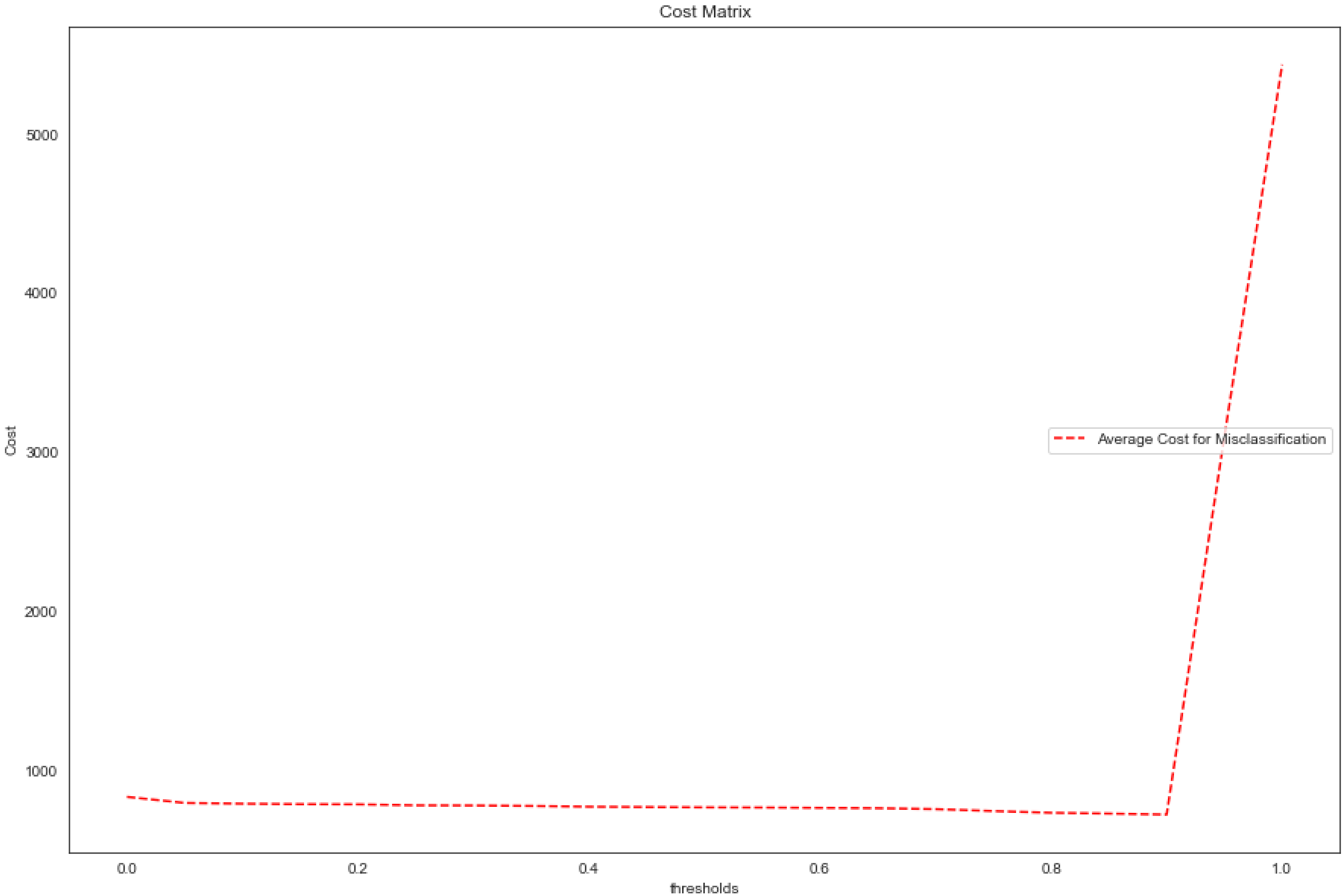
In [130]:

```
### Reset seaborn to the default background - for better viewing
sns.set_style("white")

## Create a new figure to plot
plt.figure(figsize= (15, 10))

lw = 2


## Plot the model's Precision Recall Curve
plt.plot(precision_score, recall_score, label='Precision-Recall curve (area = {})'.format(round(avg_precision, 4)))

## Put in a line to demonstrate blind luck
plt.plot([0, 1], [0, 1], color = 'black', linestyle = '--', label = 'Luck')

## Set the limits so they start at zero
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])

## Set labels for x and y
plt.xlabel('Recall')
plt.ylabel('Precision')

## Set a title and legend
plt.title('ROC Curve')
plt.legend(loc = 'lower right')

## Show the curve!
plt.show()
```



Interesting! The classifier behaved pretty similarly to the one generated above but the precision-recall curve is wildly different, and converges very strangely. I would be extremely cautious using this model.

In [132]:
```
index = 0

for t in thresholds:

    predict_thre = np.where(probas_ > t, 1, 0)  ## prediction based on the preset threshold
    clf_matrix = confusion_matrix(y_test, predict_thre)
    Cost_List[index] = clf_matrix[0][0]*cost_matrix[0][0] +clf_matrix[0][1]*cost_matrix[0][1] +clf_matrix[1][0]*cost_ma
trix[1][0] +clf_matrix[1][1]*cost_matrix[1][1]
    index+=1

## Set the figure size
plt.figure(figsize= (15, 10))

## Plot each Cost Line individually
plt.plot(thresholds, Cost_List, 'r--', label = "Average Cost for Misclassification")

## Give some labels
plt.xlabel("thresholds")
plt.ylabel("Cost")

## Title and Legend
plt.title("Cost Matrix")
plt.legend(loc = 'right')

## Show the Cost Matrix Analysis
plt.show()
```



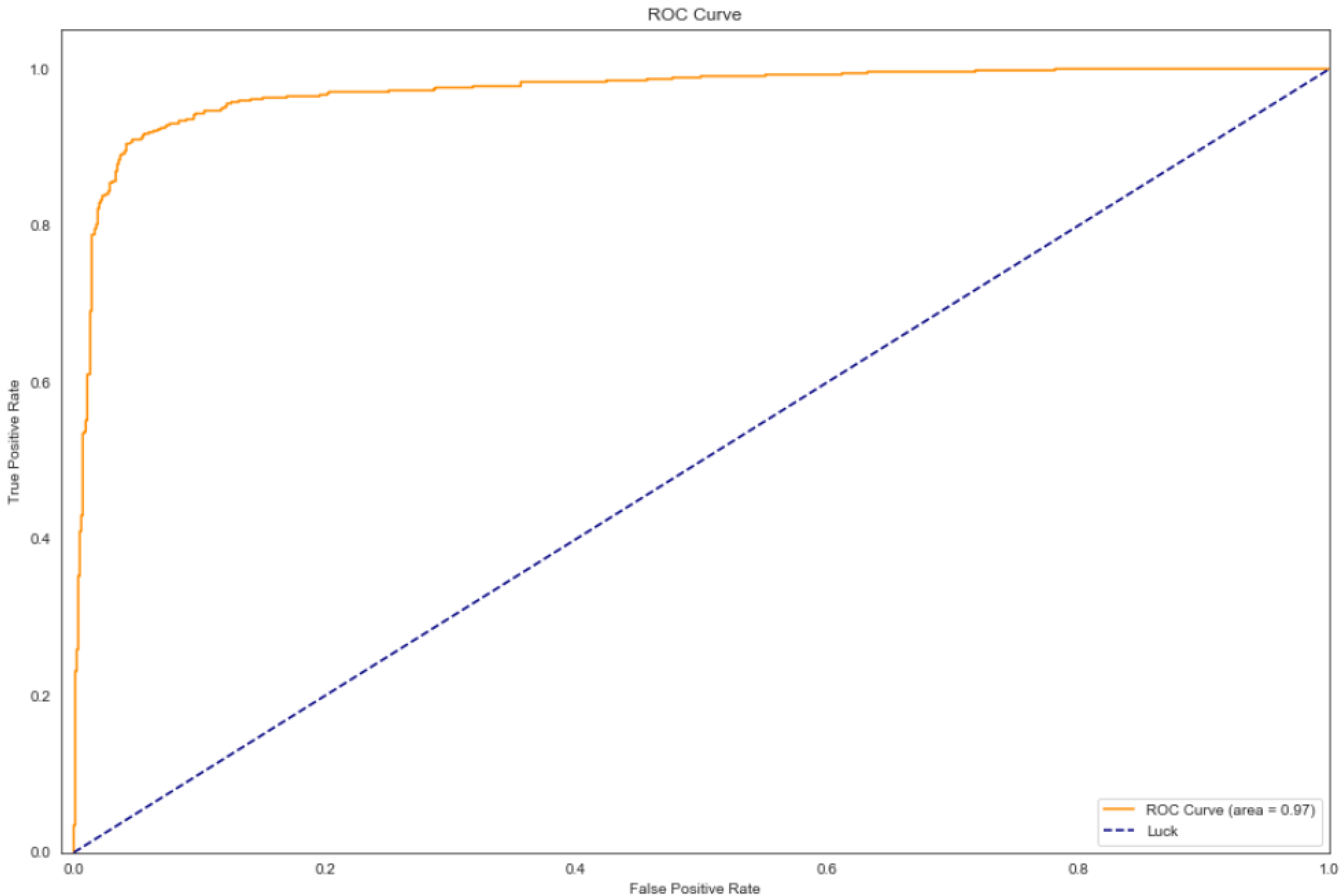
The average cost of misclassification is pretty similar to our first Keras model.

D. Conclusions / Model Evaluation

Based on the analysis completed, my selection for the best identifier for this classification would be using a Support Vector Machine with these hyper parameters tuned:

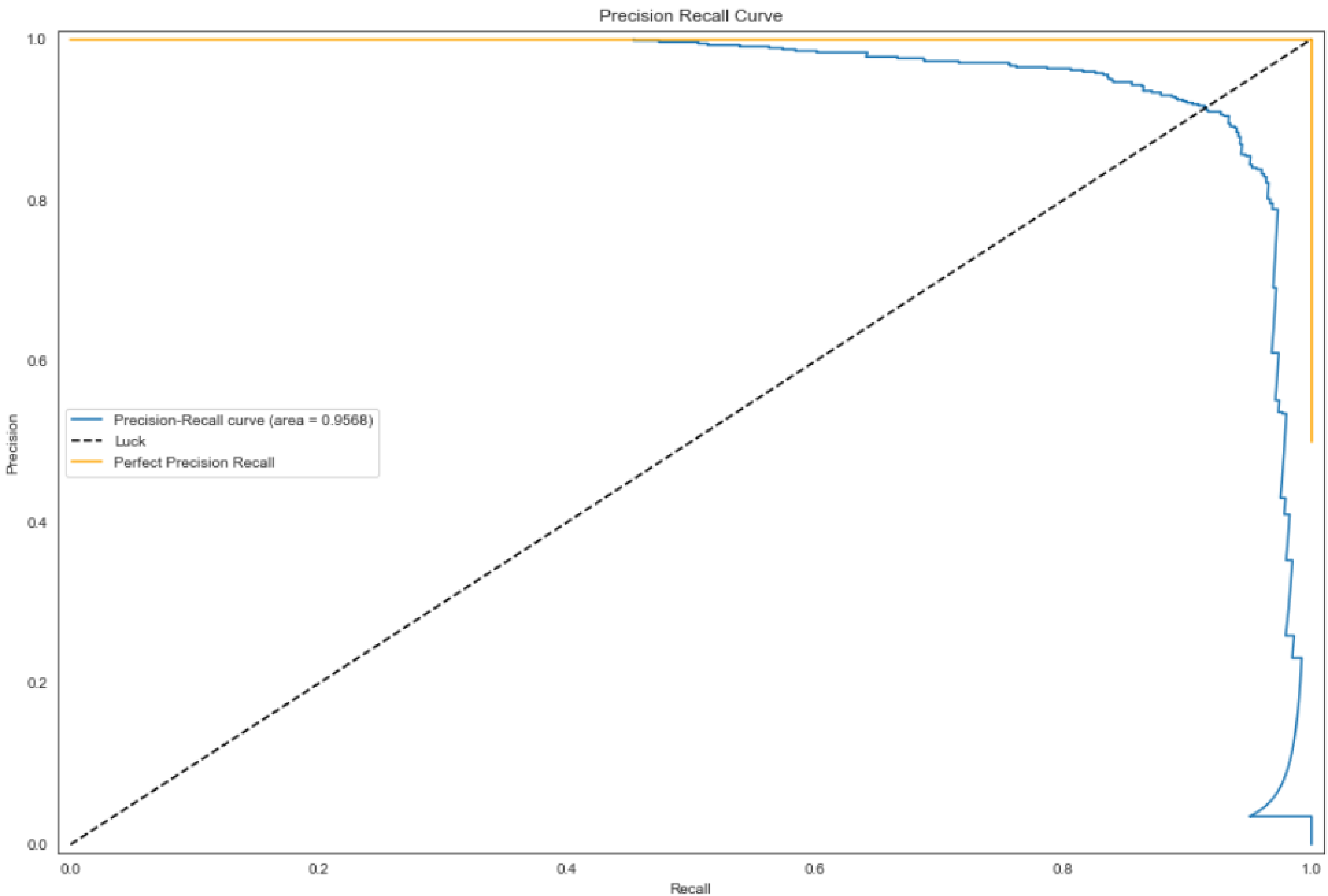
```
## Get our best params and their scores
print(svclass.best_params_)
print()

## Print out how well it performed using the best params
print(svclass.best_score_)

## save our best params so we can use them in our actual SVC model!
best_svc_params = svclass.best_params_
```

```
{'C': 1000, 'gamma': 0.0001, 'kernel': 'rbf'}

0.9347826086956522
```



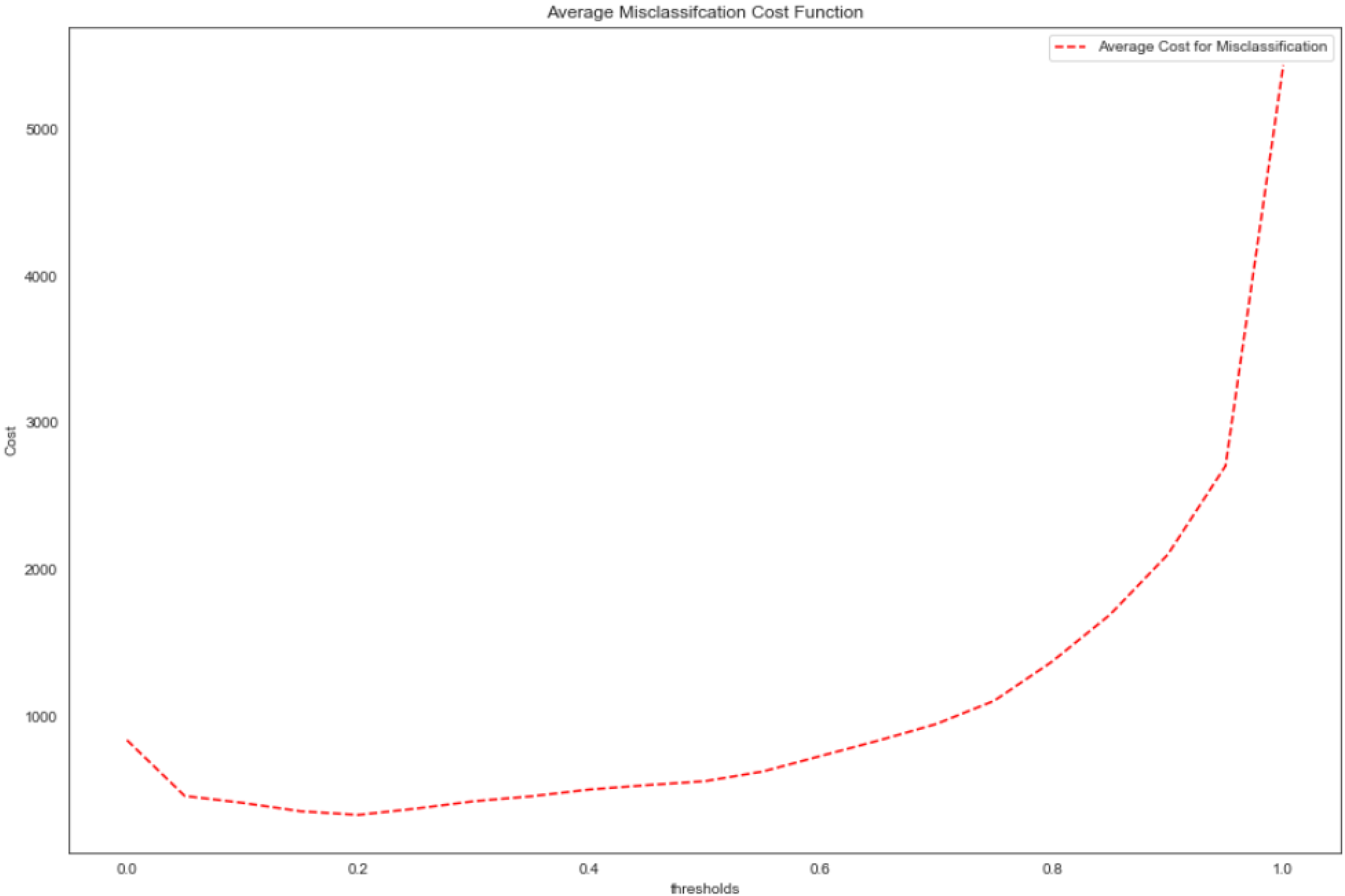
ROC Curve
1.0
0.8
0.6
0.4
0.2
0.0
True Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
False Positive Rate
ROC Curve (area = 0.97)
Luck

As we are dealing with unbalanced classes, this metric is not enough to make sure the model is accurate. That's where a precision recall curve helps accurately determine model performance.



Precision Recall Curve
Precision-Recall curve (area = 0.9568)
Luck
Perfect Precision Recall
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0

With the model's performance, as compared to random luck / guessing, and being a perfect world, it does very well. When you compare this Precision-Recall curves to the ones generated by the classifier we created with Keras, it is a much better performing classifier.


Average Misclassifcation Cost Function
Average Cost for Misclassification
Cost
5000
4000
3000
2000
1000
0.0
0.2
0.4
0.6
0.8
1.0
thresholds

As it gets closer to identifying spam, the "preferred" Support Vector Machine classifier punishes mistakes more harshly, which is the behavior that I was hoping for. Again, the idea is to punish misidentifying potential spam as not spam, so we want the model to be accurate here.

```
## Create a report to show our precision(accuracy), recall, and f1 for predictions
report = classification_report(y_test, predicted)
print(report)
```

```
[[798  39]
 [ 51 493]]

              precision    recall  f1-score   support

           0       0.94      0.95      0.95       837
           1       0.93      0.91      0.92       544

    accuracy                           0.93      1381
   macro avg       0.93      0.93      0.93      1381
weighted avg       0.93      0.93      0.93      1381
```

The support vector machine does well at both classification tasks, and might be improved with additional data to learn and refine its search.

Thank you for reading my analysis!