Case Study VII

Randy Kim, Kati Schuerger, Will Sherman
December 5, 2022

1 Introduction

Objective

The goal for this client was to generate a model that minimizes financial losses. The details of this study are to generate a classification model which can classify a binary case of either 0 or 1. Incorrect classification leads to a financial loss, with a false positive (FP)—misclassifying a 0 as a 1—leading to a loss of 20 dollars; a false negative (FN)—misclassifying a 1 as a 0—causing a loss of 100 dollars.

The final deliverable for this case would be our final neural network architecture as an ensemble of the three baseline models: stochastic gradient descent classifier, random forest classifier, and dense neural network classifier. Additionally, we (would) provide the weight matrices for the first dense neural network and the ensemble neural network.

2 Methods

Data

The data to build the model were provided by the client. The target variable was binary: 1 or 0. The data records consisted of 50 features: 47 numeric and 3 categorical. The client didn't provide any metadata on the features for analysis; therefore, not much other information is available on the features in or the subject matter of the data. This does not inhibit the ability to produce a useful model, it just meant that there are limited options to use domain knowledge to improve the model.

Data pre-processing (EDA)

Because the client was not interested in understanding feature importance in modeling, we opted for less explainable models and methods. PCA was investigated as an option but was ultimately discarded as it did not yield improved performance in any of the models investigated.

Imputation and Missing Values

The client's data included missing values in every single feature excluding the target. However, there were very few missing within each attribute: the maximum percentage missing for any single data column was 0.03%. For all the numeric features, we investigated their distributions and found them to be Gaussian; therefore, we opted to impute missing values for these features as the median of the distribution.

For the categorical data columns, we decided it made more sense to drop these particular records without more information on them. This represented an information loss of 0.089%.

One-hot-encoding

Initial investigation of the data identified 5 features which were initially assumed to be non-numeric data (x24, x29, x30, x32, and x37). However, two of these features, x32 and x37, actually represented numeric data with additional string characters (*e.g.*, $ and %). These two were transformed into numeric features. The remaining categorical features were one-hot-encoded prior to modeling. Once these new features were added, the original categorical vectors were dropped from the data.

Normalize data

One step we took to remove bias from the data was normalization. A normalization scheme takes the value ranges of each feature and fits them to a common scale. This is done in each column by taking each value (x) and subtracting by the mean of the column (μ), then dividing by the standard deviation of the column (σ):

$$z = \frac{x - \mu}{\sigma}$$

We utilized SKLearn's StandardScaler to do the normalization.

This approach is exceedingly important for certain models such as neural networks. Neural networks solve the given task by evaluating and updating weights and slope values for each feature as it passes data through the network.

Model Types

We explored several model types to identify the method to deliver a useful model to the client. Models evaluated include random forest, logistic regression, neural network (sequential), stochastic gradient descent (SGD), and support vector machine (SVM).

Neural network structure

For the Neural Network model development, TensorFlow and the Keras API were used. To build the network, the specifications for the structure need to be provided. Next, the model must be compiled. Then the model can be fit and provide predictions.

Structure specifications

The first piece of the network structure is the input layer. This is the number of features that will be passed into the model. After feature engineering, there were 67 inputs.

The second piece of the network is hidden layers—in this case, dense layers. These are the worker nodes that process the data to identify appropriate slopes and weights for each feature. This study used the sequential method (TensorFlow Keras), each layer implemented in order, without skipping any layers.

An activation function must also be specified for each layer of the network. The activation function must be appropriate for the task (classification); if an incorrect activation function is provided, the network will not fail, it will try to solve the model using whatever activation function is given. For this reason, it is important to pick an appropriate function so that the results will be aligned with what we expect. Activation functions applied included: ReLu, Tanh, and sigmoid.

Finally, the last layer of the network is the output layer, which contains the outputs of the model (predictions).[1]

Model compilation & fitting

The step for compiling the model included specifying an optimizer, a loss function, and metrics for evaluation of performance on the validate set during training. The optimizer was used by the model to optimize the loss function and to update the weight and bias matrices. Tested network models used the *AdaDelta* and *Adam* optimizer.

In this model, the loss function evaluated over our training epochs to optimize the neural network was *binary cross entropy*. Metrics are additional measures of performance to help the network as it is learning on the training and validation data. Metrics included were accuracy, precision, recall, and the custom F1 score.

The model was fit on training data with a batch size of 10,000 records. The model was permitted to run through as many as 1,000 epochs; however, with early stopping criteria measured on the loss with a minimum delta of 1×10^{-5}, training typically finished between 15 and 30 epochs.

Early stopping criteria allowed the model to monitor the validation loss and mathematically determine when that loss had stopped improving.

Train/validate/test split

All model types can suffer from overfitting. To combat this, the data were split into three sections: train, validate, test, prior to model fitting. The train set and validate set were then used to build the model; validation loss was monitored to assess when the model has stopped learning, as determined by early stopping criteria. The test set was then used to evaluate model performance on unseen data. This was done with SKLearn's train_test_split.

Due to the unbalanced representation of targets (Figure 1) within the data records (*i.e.,* 0s and 1s), we implemented a stratified random sampling approach to ensure we got a balanced representation in our train, validate, and test splits. The distribution of the split was 80/12/8, to have a large enough validation set (which would be used as the training set of the ensemble model) as well as a robust final test set.

Model evaluation & Scoring metrics

Several methods were used to evaluate model performance: binary cross-entropy, accuracy, precision, recall, confusion matrix, receiver operating characteristic area under the curve (ROC AUC), and a custom F1 score (additional details below). Cross-entropy is a loss function that can be used in TensorFlow to solve classification tasks. Binary cross-entropy is specific to tasks where the target value only has two possible labels, as in this case (0 and 1). This loss is used by the network to optimize model learning. The loss shows how well a model works in terms of prediction error. If the predictions are close to the actuals, the loss will be minimum and if the predictions are

[1] Structure specifications text adapted from "Case Study VI" (Kim, Schuerger, Sherman).

away from the actuals, the loss value will maximum. For this case study, we used Binary Cross Entropy which compares each prediction and actual that is either 0 or 1.

Accuracy, precision, and recall were used to evaluate model training for the neural network and SGD classifier. ROC AUC were used to evaluate performance for the random forest classifier. Accuracy alone can be misleading if used with imbalanced data. Therefore, we focused on F1 score; this is also the most likely metric to be of interest to the client.

We also utilized a confusion matrix for model evaluation as it gives a better understanding of how the model is performing.

F1 Score

The client advised that a false negative results in a higher financial loss than a false positive. A false negative costs $100; a false positive costs $20.

A typical F1 score is a measure of the accuracy that better takes these into account. F1 score represents the harmonic mean of precision and recall metrics, and attributes equal weight to both (50/50).

Precision

Within everything that has been predicted as a positive, precision counts the percentage that is correct. A precise model may not find all the positives, but the ones that the model does classify as positive are very likely to be correct.

Precision is calculated as true positives divided by total predicted positives (which includes predicted positives correctly identified, and predicted positives that are true negative – value 0).

$$Precision = \frac{\#\ of\ True\ Positives}{\#\ of\ True\ Positives + \#\ of\ False\ Positives}$$

Recall

Recall is calculated as true positives divided by true positives (that our model predicted correctly) plus false negatives (model predicted negative, but true/actual value was positive).

$$Recall = \frac{\#\ of\ True\ Positives}{\#\ of\ True\ Positives + \#\ of\ False\ Negatives}$$

An ideal model would have the best of both metrics, however, in real life, we deal with the Precision-Recall tradeoff. The model can be adjusted to increase precision at a cost of a lower recall. (See appendix B for additional details.)

3 Results

The client's data consisted of a distribution of classes (*i.e.,* 0 and 1) which were slightly imbalanced (Figure 1).

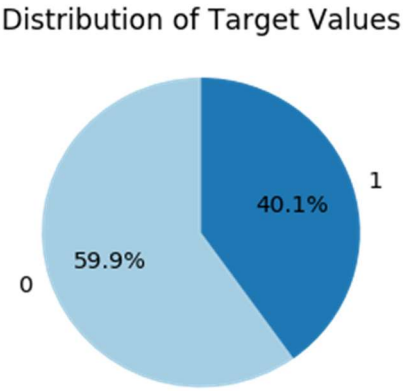

Figure 1. Percent distribution for the number of records where the target was 0 or 1.

Upon initial modeling of the three baseline models, the F1 score was found to be the highest for the dense neural network (Table 1).

Model	F1 Score
SGD Classifier	0.6527
Random Forest Classifier	0.8235
Dense Neural Network	0.9276

Table 1. F1 scores for predicting both class 1 and class 0 for each of the three baseline models.

Evaluating the binary cross-entropy loss for the neural network model, the loss for both traning and validation data decrease fairly quickly, converging to an average loss of approximately 0.10 and 0.15, respectively (Figure 2a). However, the trend in the validation data indicates that as the number of epoch increases beyond 30 to 60 the variance in loss increases without appreciable

decrease in the loss value. The accuracy between training and validation data exhibits similar behavior and ranges between 93 percent to 96 percent (Figure 2b).

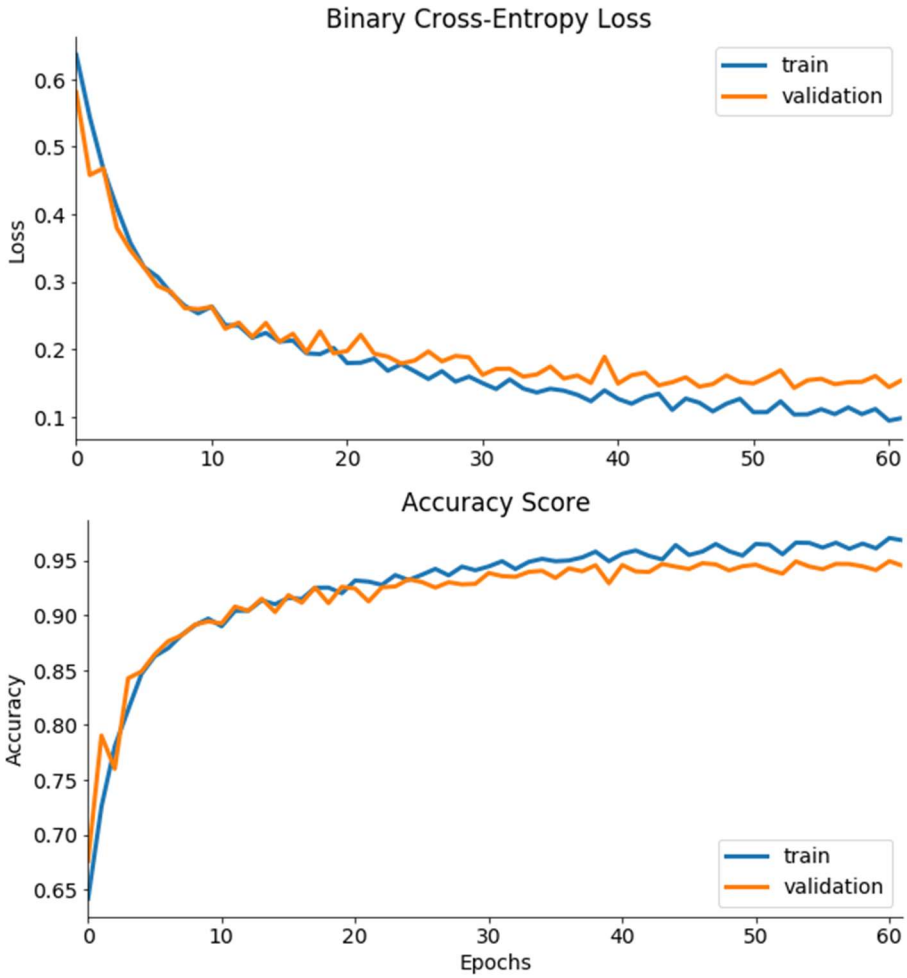

Figure 2. a) Binary cross-entropy loss as a function of the number of epochs; plateau reached after (typically) between 40 and 60 epochs. b) Tracking accuracy score for training and validation datasets as a function of the number of epochs; plateau for validation accuracy indicates no further improvements with increased epochs.

One further investigation we made, initially, was on the distribution of the miss-classified predictions for the neural network. We found that the distribution of misclassified probabilities for the neural network had peaks at probabilities 0.000 and 0.999 (Figure 3a). Ideally, we'd prefer to have higher rates of misclassification further away from our certainties of 0 and 1. We looked at the histogram distributions of miss-classifications for the other two models. They had much nicer error distributions: more normal and more centered on 0.5000 (Figures 3b and 3c).

This drove our decision to attempt an ensemble model: to utilize the higher certainty at the extremes of the distribution provided by the random forest and SGD classifiers.

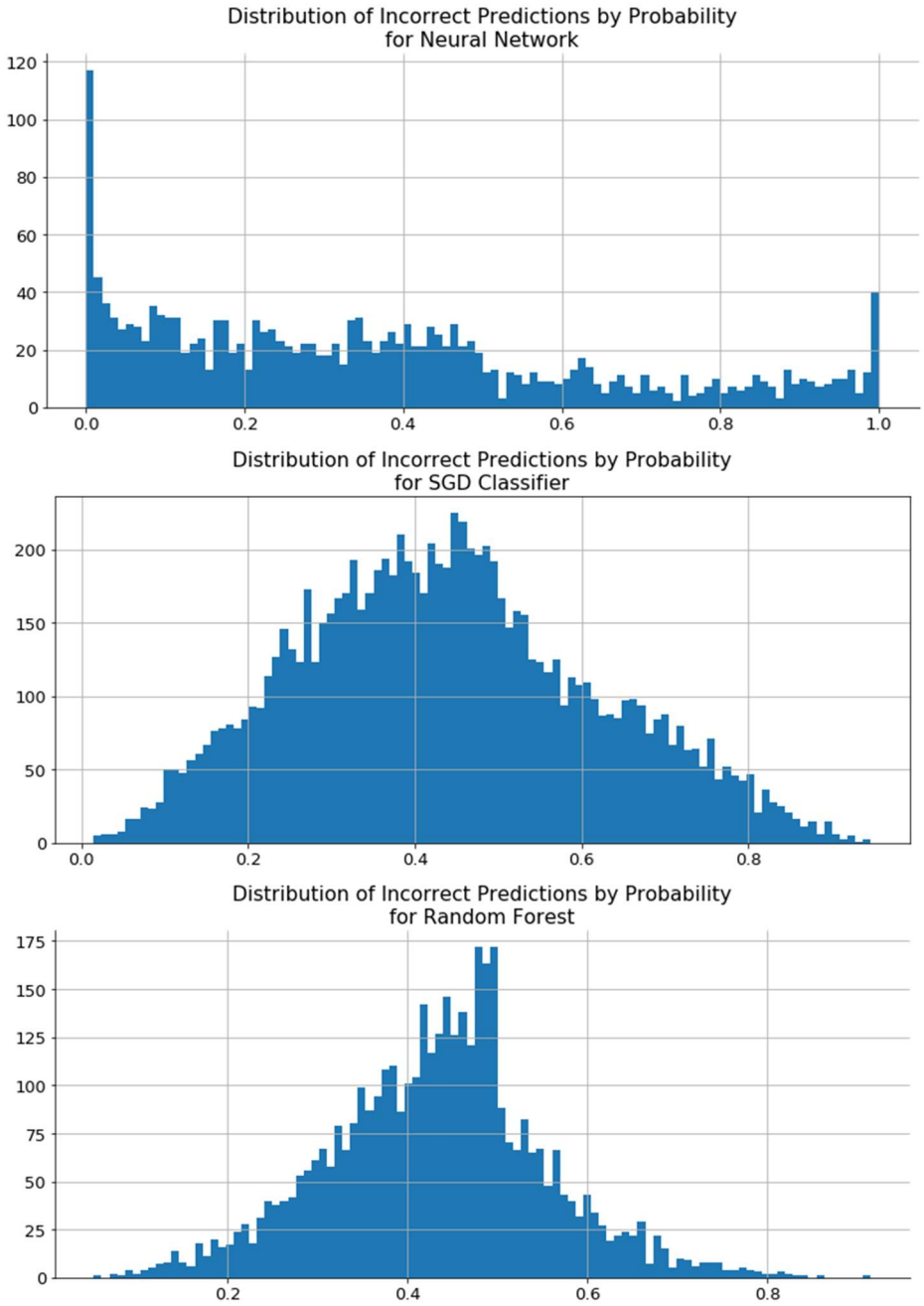

Figure 3. Histograms of probability assignment for each validation record to either class 0 or class 1 for (a) the neural network, (b) the SGD classifier, and (c) the random forest classifier.

The final neural network (NN) was able to achieve an almost 1% increase in F1 score (Table 2).

Model	F1 Score
Top Performing Base Model (NN)	0.9276
Ensemble Model (NN)	0.9317

Table 2. F1 scores for predicting class 1 and class 0 for the best-performing baseline model and the ensemble neural network (NN).

To further evaluate our model, we have a confusion matrix (Figure 4) which details the performance of the model in terms of the True label vs Predicted label. We can see that number of misclassified "1s" was 456 out of the 12,789 test instances; this represents a false negative rate of around 8%. The number of misclassified "0s" was 229 compared to 7,429 correct classifications. This represents a false positive rate of roughly 3%.

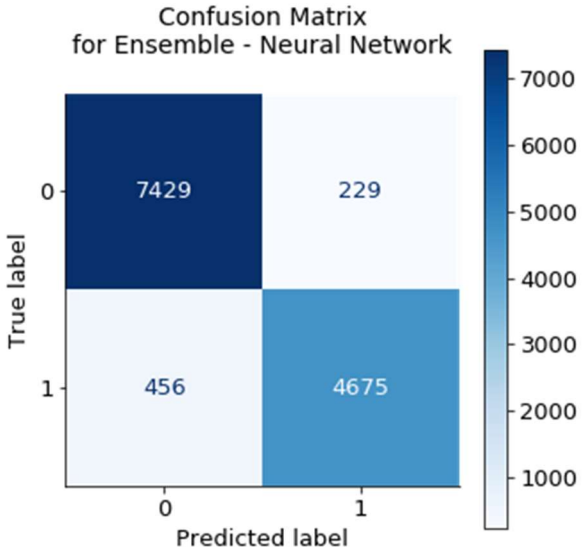

Figure 4. Confusion matrix on the holdout data for the ensembled neural network.

We then re-evaluated the distribution of classification probabilities. Unfortunately, we did not achieve a more normal distribution. However, we did cut the misclassification rate in half for probabilities close to 1.00 and to nearly a third of the starting value for those close to 0.00. This still represents an improvement in the distribution of misclassifications.

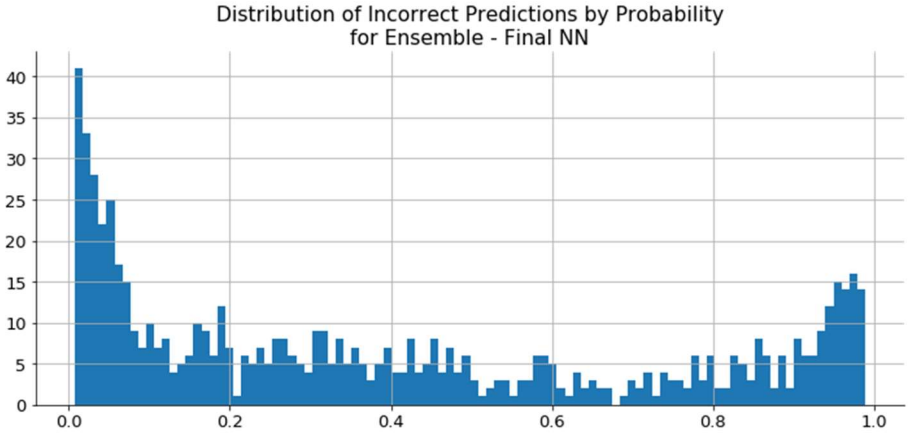

Figure 5. Histograms of probability assignment for each validation record to either class 0 or class 1 for the final model (ensemble neural network).

4 Conclusion

The goal of this case study was to create a model that could minimize financial losses with high accuracy. We believe that dense neural networks came out with the best result because we are dealing with a large set of data and the learning approach of dense neural networks has advantages of discovering complex relationships within data through its algorithms and many-layered structure.

As discussed above, the dense neural network model produced the highest F1 score for a single model which was 0.9276 while random forest classifier and SGD classifier produced F1 scores of 0.8235 and 0.6527 respectively. But with the ensemble, overall loss analysis shows that the client is likely to miss sales in 5% of cases. The total dollar loss per sale has been reduced to an average of $3.92/sale.

Appendix A

References

https://towardsdatascience.com/the-f1-score-bec2bbc38aa6

Code

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:

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

from sklearn.model_selection import train_test_split, KFold
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV
from sklearn.calibration import CalibratedClassifierCV

from sklearn.linear_model import SGDClassifier
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix, f1_score

# In[2]:

df = pd.read_csv('final_project.csv')

# In[3]:
```

```
df.head()

# In[4]:

df.info()

# In[5]:

pd.set_option('display.max_columns', None)
df.describe()

# In[6]:

df.isnull().sum().sum()

# In[7]:

colname = list(df.columns)
rank = {}

for i in range(len(colname)):
    count = df[df.columns[i]].isna().sum()
    rank[i] = count

    if count > 0:
        print("Column '{col}' has {ct} NAs".format(col = colname[i], ct = count))

# In[8]:

column_rank = pd.DataFrame(rank.items(), columns = ['Attr', 'Missing Count'])
column_rank["Missing Percentage"] = round(column_rank["Missing Count"]/len(df)*100,2)
column_rank.sort_values("Missing Count", ascending=False)

# In[9]:

num_vec = list(range(df.shape[1]))
non_num = [24,29,30,32,37,50]
num_vec = [x for x in num_vec if x not in non_num]

# In[10]:

df.iloc[:,num_vec]

# In[11]:

plot_df = df.iloc[:,num_vec]
plot_df.hist(bins=100, figsize = (20,15))
plt.show()

# In[12]:

df = df.fillna(df.median())

# In[13]:

for i in range(len(colname)):
    count = df[df.columns[i]].isna().sum()
    rank[i] = count
```

```
    if count > 0:
        print("Column '{col}' has {ct} NAs".format(col = colname[i], ct = count))

# In[14]:

df.dropna(axis=0, inplace=True)

# In[15]:

df_cleaned = pd.DataFrame.copy(df)
df_cleaned = df_cleaned.reset_index(drop=True)

# In[16]:

df_cleaned['y'].value_counts()

# In[17]:

df_cleaned.shape

# In[18]:

valueCounts = df_cleaned['y'].value_counts()
color = sns.color_palette('Paired')

plt.figure(figsize=(4, 4), dpi=80)
plt.pie(valueCounts, colors = color, autopct='%.01f%%', labels = ['0', '1'], startangle = 90)
plt.title("Distribution of Target Values")
plt.show()

# In[19]:

df_cleaned.select_dtypes(include=[object])

# In[20]:

df_cleaned['x37'] = df_cleaned['x37'].map(lambda x: x.lstrip('$')).astype('float')
df_cleaned['x37']

# In[21]:

df_cleaned['x32'] = df_cleaned['x32'].map(lambda x: x.rstrip('%')).astype('float')
df_cleaned['x32']

# In[22]:

plot_df = df_cleaned.iloc[:,[32,37]]
plot_df.hist(bins=100, figsize = (5,2))
plt.show()

# In[23]:

### num_vec is used to process x-data using standard-scaler later

num_vec = df_cleaned.select_dtypes(exclude=[object]).columns.tolist()
to_go = num_vec.index('y')
num_vec.pop(to_go)
```

```
# In[24]:

df_cleaned = pd.get_dummies(df_cleaned, columns=['x24', 'x29', 'x30'], prefix=['x24', 'x29', 'x30'])
df_cleaned

# In[25]:

df_cleaned.isnull().sum().sum()

# In[26]:

# features 2 and 6 have near-perfect correlation
# features 38 and 41 have near-perfect correlation
plt.figure(figsize=(20,15))

heatmap = sns.heatmap(df_cleaned.corr(), vmin=-1, vmax=1, annot=False, cmap = 'Blues')
heatmap.set_title('Corrleation Heatmap', fontdict={'fontsize':12}, pad=12)

# In[27]:

# No Good Distribution from Quadratic
fig, axs = plt.subplots(nrows=int(len(num_vec)/5)+1, ncols=2, figsize=(20,15))
plt.subplots_adjust(hspace=0.4)
fig.suptitle('Quadratic Transformation', fontsize=14, y=0.95)

for feature, ax in zip(num_vec, axs.ravel()):
    x_plot = df_cleaned[feature]
    plotter = df_cleaned[feature]**2
#     ax = fig.add_subplot(5,5,ax)
    ax.scatter(x_plot, plotter, c=df_cleaned['y'])
    ax.set_title(feature)

plt.tight_layout()
plt.show()

# In[28]:

df_cleaned.iloc[:,~df_cleaned.columns.isin(num_vec)]

# In[29]:

df_cleaned.columns[48:68]

# In[30]:

df_cleaned.shape

# In[31]:

x_Data = df_cleaned.loc[:,df_cleaned.columns!='y']
y_Data = df_cleaned.loc[:,'y']
x_Data.shape

# In[32]:

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
# In[33]:

def scale_just_nums(dataframe, vector):
    only_x = dataframe.loc[:,vector]
    scaled_x = pd.DataFrame(scaler.fit_transform(only_x))
    all_x_data = pd.concat([scaled_x.reset_index(drop=True),
                            dataframe.loc[:,~dataframe.columns.isin(vector)].reset_index(drop=True)],
                           axis=1)
    all_x_data = pd.DataFrame(all_x_data)

    return all_x_data

# In[34]:
# Set up Train/Test Split for Modeling

x_train_raw, x_test_raw, y_train, y_test = train_test_split(
    x_Data, y_Data, test_size=0.2, stratify=y_Data, random_state=775)

x_test1_raw, x_test2_raw, y_test1, y_test2 = train_test_split(
    x_test_raw, y_test, test_size=0.4, stratify=y_test, random_state=543)

# In[35]:

# run the standard scaler on the x_data:  train / test_1 / test_2
x_train = scale_just_nums(x_train_raw, num_vec)

# In[36]:

x_train.describe()

# In[37]:

x_train.shape

# In[38]:

from sklearn.decomposition import PCA
pca = PCA()
x_train_pca = pd.DataFrame(pca.fit_transform(x_train))

exp_var_pca = pca.explained_variance_ratio_
#
# Cumulative sum of eigenvalues; This will be used to create step plot
# for visualizing the variance explained by each principal component.
#
cum_sum_eigenvalues = np.cumsum(exp_var_pca)
#
# Create the visualization plot
#
plt.figure(figsize=(8,6))
plt.bar(range(0,len(exp_var_pca)), exp_var_pca, alpha=0.5, align='center', label='Individual explained variance')
plt.step(range(0,len(cum_sum_eigenvalues)), cum_sum_eigenvalues, where='mid',label='Cumulative explained
variance', linewidth=2)
plt.axhline(0.98, 0.05, 0.6, color='red', linewidth=0.8)
plt.axvline(40, 0.01, 0.93, color='red', linewidth=0.8)
plt.ylabel('Explained variance ratio')
plt.xlabel('Principal component index')
plt.legend(loc='best')
```

```
plt.tight_layout()
plt.show()

# In[39]:

x_train_pca.iloc[:, np.arange(0,40)]

# In[40]:

x_train_pca.describe()

# #### Generate Test Data for both normal and PCA data

# In[43]:

# run the standard scaler on the x_test_raw:  train / test_1 / test_2
x_test = scale_just_nums(x_test_raw, num_vec)

# In[44]:

valueCounts = y_test.value_counts()
color = sns.color_palette('Paired')

plt.figure(figsize=(4, 4), dpi=80)
plt.pie(valueCounts, colors = color, autopct='%.01f%%', labels = ['0', '1'], startangle = 90)
plt.title("Distribution of Target Values")
plt.show()

# In[45]:

x_test_pca = pd.DataFrame(pca.fit_transform(x_test))

# #### Generate Test Data SUBSETS

# In[46]:

x_test1 = scale_just_nums(x_test1_raw, num_vec)
x_test2 = scale_just_nums(x_test2_raw, num_vec)

# In[47]:

print(x_test1.shape)
print(x_test2.shape)

# #### Generate PCA Test Data SUBSETS

# In[48]:

x_test1_pca = pd.DataFrame(pca.fit_transform(x_test1))
x_test2_pca = pd.DataFrame(pca.fit_transform(x_test2))

# #### Setting Up SGD Classifier HYPERS and CV object

# In[49]:

folder = KFold(n_splits=10, shuffle=True, random_state=867)
```

```
# In[50]:

sgd_hyps = {'loss':['hinge', 'modified_huber', 'squared_hinge', 'perceptron',
                    'huber', 'epsilon_insensitive', 'squared_epsilon_insensitive'],
            'alpha':np.logspace(-8,8,100),
            'fit_intercept':[True, False],
            'tol':np.logspace(-10,-1,8),
            'epsilon':np.logspace(-4,4,9),
            'learning_rate':['optimal','constant','invscaling','adaptive'],
            'eta0':np.logspace(-3,9,13),
            'power_t':[x + 0.5 for x in np.linspace(-10000,10000,21)],
            'validation_fraction':[0.2],
            'n_iter_no_change':[5],
            'class_weight':[None,'balanced']
}

# In[51]:

sgdc = SGDClassifier(max_iter=10000, penalty='l2', fit_intercept=True, random_state=487, n_iter_no_change=10,
verbose=20)

# #### Function for Running Either PCA or Scaled X_train and generating all outputs for SGD

# In[52]:

def sgd_any_inputs(model, hypers, cv_object, train_x, train_y):
    # initial fit
    random_search = RandomizedSearchCV(model, param_distributions=hypers, scoring='f1',
                                       n_iter=200, n_jobs=30, cv=cv_object,
                                       random_state=816, verbose=20)
    random_search.fit(train_x, train_y)

    current_best = random_search.best_estimator_
    print(current_best,'\n')
    print('f1 score:\t\t\t',random_search.best_score_)

    # tuned hypers as ranges
    tuned_hypers = {'loss':[current_best.loss],
                    'alpha':[abs(x) for x in np.random.normal(loc=current_best.alpha,
                                                              scale=2*math.exp(math.log(current_best.alpha)),
                                                              size=20).tolist()
                             ],
                    'fit_intercept':[current_best.fit_intercept],
                    'tol':[abs(x) for x in np.random.normal(loc=current_best.tol,
                                                            scale=2*math.exp(math.log(current_best.tol)),
                                                            size=10).tolist()
                           ],
                    'eta0':[current_best.eta0],
                    'epsilon':[current_best.epsilon],
                    'learning_rate':[current_best.learning_rate],
                    'power_t':[current_best.power_t],
                    'validation_fraction':[0.2],
                    'n_iter_no_change':[5],
                    'class_weight':[current_best.class_weight]
                   }

    # secondary fit
    grid_search = GridSearchCV(model, param_grid=tuned_hypers, scoring='f1',
```

```
                        n_jobs=32, cv=cv_object, verbose=20)
    grid_search.fit(train_x, train_y)

    final_best = grid_search.best_estimator_
    print(final_best,'\n')
    print('f1 score:\t\t\t',grid_search.best_score_)

    # calibrated CV for losses (not log or modified_huber)
    calibrated_clf = CalibratedClassifierCV(base_estimator=model, method='sigmoid', cv=5)  # set the SGD classifier
as the base estimator

    cal_params = {'base_estimator__alpha':[ # note 'base_estimator__' in the params because you want to change
params in the SGDClassifier
        abs(x) for x in np.random.normal(loc=current_best.alpha,
                                         scale=2*math.exp(math.log(current_best.alpha)),
                                         size=20).tolist()
    ]}
    cal_search = GridSearchCV(estimator=calibrated_clf, param_grid=cal_params, cv=5)
    cal_search.fit(train_x, train_y)

    cal_best = calibrated_clf.set_params(**cal_search.best_params_)

    return final_best, cal_best

# #### SGD with PCA

# In[53]:

# sgd_pca_model, = sgd_any_inputs(sgdc, sgd_hyps, folder, x_train_pca.iloc[:, np.arange(0,40)], y_train)
# sgd_pca_model

# In[54]:

# pca_preds = sgd_pca_model.predict(x_test_pca.iloc[:, np.arange(0,40)])
# f1_score(y_test, pca_preds)

# #### SGD with NO PCA

# In[55]:

sgd_model,cal_model = sgd_any_inputs(sgdc, sgd_hyps, folder, x_train, y_train)
sgd_model

# In[56]:

sgd_preds = sgd_model.predict(x_test)
f1_score(y_test, sgd_preds)

# In[57]:

cal_model.fit(x_train, y_train)

# In[58]:

sgd_pred_probs = cal_model.predict_proba(x_test)
sgd_pred_probs
```

```
# In[59]:

font = {'size'   : 13}
plt.rc('font', **font)

cm = confusion_matrix(y_test, cal_model.predict(x_test))
c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray([0,1]))

fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor Calibrated Model for SGD Classifier\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='')

# In[60]:

font = {'size'   : 13}
plt.rc('font', **font)

cm = confusion_matrix(y_test, sgd_preds)
c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray([0,1]))

fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor SGD classifier\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='')

# In[61]:

sgd_preds = cal_model.predict(x_test)

# In[62]:

sgd_pred_list = sgd_preds.tolist()
sgd_plot = pd.DataFrame({'actual':list(y_test),
                         'predictions':sgd_pred_list})
sgd_plot

# In[63]:

sgd_probs_vector = sgd_pred_probs[:,1]
print(sgd_probs_vector.shape)


# In[64]:

sgd_plot['comparison'] = np.where(sgd_plot['actual']==sgd_plot['predictions'], True, False)
sgd_plot.head(15)

# In[65]:

pd.DataFrame(sgd_probs_vector[sgd_plot['comparison']==False]).hist(bins=100, figsize = (12,5))
plt.title('Distribution of Incorrect Predictions by Probability\nfor SGD Classifier')
plt.show()

# #### Trying NN on non-PCA data

# In[66]:
```

```
import tensorflow as tf
from tensorflow.keras.optimizers import Adam
from matplotlib import rcParams
from tensorflow.keras.callbacks import EarlyStopping
import seaborn as sns

# In[67]:

# tf.random.set_seed(12)

model = tf.keras.Sequential([
    tf.keras.layers.Dense(512, input_shape=(67,), activation='relu'),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dense(256, activation='tanh'),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# In[69]:

model.compile(
    optimizer='adadelta',
    loss='binary_crossentropy',
    metrics=[
        tf.keras.metrics.BinaryAccuracy(name='accuracy'),
        tf.keras.metrics.Precision(name='precision'),
        tf.keras.metrics.Recall(name='recall'),
    ]
)

# In[70]:

model.summary()

# #### NN fit on non-PCA data

# In[71]:

stops = EarlyStopping(monitor='val_loss', patience=8, min_delta=1e-6)

model_fit = model.fit(x_train, y_train,
                      validation_data=(x_test1, y_test1),
                      epochs=1000,
                      batch_size=10000,
                      callbacks=[stops])

# In[72]:

rcParams['figure.figsize'] = (10, 5)
rcParams['lines.linewidth'] = 3
rcParams['font.size']=14
rcParams['axes.spines.top'] = False
rcParams['axes.spines.right'] = False

# In[73]:

history_df = pd.DataFrame(model_fit.history)
history_df[['loss', 'val_loss']].plot()
```

```
plt.title('Binary Cross-Entropy Loss')
plt.ylabel('Loss')
plt.legend(['train','validation'])

history_df = pd.DataFrame(model_fit.history)
history_df[['accuracy', 'val_accuracy']].plot()
plt.title('Accuracy Score')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')
plt.legend(['train','validation'], loc='lower right')

# In[74]:

nn_preds = model.predict_classes(x_test)

# In[75]:

print(y_test.shape)
print(nn_preds.shape)

# In[76]:

f1_score(y_test, nn_preds) # evaluating on all “test” values

# In[77]:

font = {'size'   : 13}
plt.rc('font', **font)

cm = confusion_matrix(y_test, nn_preds)
c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray([0,1]))

fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor Neural Network\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='')

# In[78]:

nn_pred_probs = model.predict(x_test)
nn_pred_probs

# In[79]:

nn_pred_list = [item for sublist in nn_preds.tolist() for item in sublist]

# In[80]:

nn_plot = pd.DataFrame({'actual':list(y_test),
                        'predictions':nn_pred_list})
nn_plot

# In[81]:

nn_plot['actual'][0] == nn_plot['predictions'][0]

# In[82]:
```

```
nn_plot['comparison'] = np.where(nn_plot['actual']==nn_plot['predictions'], True, False)
nn_plot.head(15)

# In[83]:

pd.DataFrame(nn_pred_probs[nn_plot['comparison']==False]).hist(bins=100, figsize = (12,5))
plt.title('Distribution of Incorrect Predictions by Probability\nfor Neural Network')
plt.show()

# In[84]:

nn_plot['comparison'].value_counts()

# #### NN fit on PCA data

# In[85]:

# stops = EarlyStopping(monitor='val_loss', patience=8, min_delta=1e-6)

# model_fit = model.fit(x_train_pca, y_train,
#                       validation_data=(x_test1_pca, y_test1),
#                       epochs=1000,
#                       batch_size=10000,
#                       callbacks=[stops])

# In[86]:

# history_df = pd.DataFrame(model_fit.history)
# history_df[['loss', 'val_loss']].plot()
# plt.title('Binary Cross-Entropy Loss')
# plt.ylabel('Loss')
# plt.legend(['train','validation'])

# history_df = pd.DataFrame(model_fit.history)
# history_df[['accuracy', 'val_accuracy']].plot()
# plt.title('Accuracy Score')
# plt.ylabel('Accuracy')
# plt.xlabel('Epochs')
# plt.legend(['train','validation'], loc='lower right')

# In[87]:

# f1_score(y_test2, nn_preds)

# In[88]:

# font = {'size'   : 13}
# plt.rc('font', **font)

# cm = confusion_matrix(y_test2, nn_preds)
# c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray([0,1]))

# fig, ax = plt.subplots(figsize=(5,5))
# ax.grid(False)
# plt.title('Confusion Matrix\nfor Neural Network\n', fontsize=14)
# c_disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='')
```

```
# #### Random Forest Classifier

# In[89]:

rf_hyps = {
   'criterion': ['gini'],
   'max_depth': range(2,11,2),
   'max_features': ['sqrt'],
   'min_impurity_decrease': np.logspace(-10,2,13),
   'class_weight': [None, 'balanced'],
   'n_jobs':[10]
}

# In[90]:

rf_clf = RandomForestClassifier()
folder = KFold(n_splits=10, shuffle=True, random_state=867)

# In[91]:

rf_search = RandomizedSearchCV(estimator=rf_clf,
                               param_distributions=rf_hyps,
                               n_jobs=30,
                               cv=folder,
                               scoring='roc_auc',
                               refit='roc_auc',
                               n_iter=20,
                               verbose=5)

# In[92]:

result_rf = rf_search.fit(x_train, y_train)

# In[93]:

print(result_rf.best_params_)
print(result_rf.best_estimator_)
print('f1 score:\t\t\t',result_rf.best_score_)
print('time to fit individual model:\t',result_rf.cv_results_['mean_score_time'].sum().round(2))
best_rf = result_rf.best_estimator_

# In[94]:

rf_preds = best_rf.predict(x_test)
f1_score(y_test, rf_preds)

# In[95]:

cm = confusion_matrix(y_test, rf_preds)

font = {'size'   : 13}
plt.rc('font', **font)

c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray(['deny','allow']))
fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor Random Forest classifier\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='')
```

```
# In[96]:

rf_probas = best_rf.predict_proba(x_test)[:,1]

# In[97]:

rf_plot = pd.DataFrame({'actual':list(y_test),
                        'predictions':rf_preds})
rf_plot['comparison'] = np.where(rf_plot['actual']==rf_plot['predictions'], True, False)
rf_plot.head(15)

# In[98]:

pd.DataFrame(rf_probas[rf_plot['comparison']==False]).hist(bins=100, figsize = (12,5))
plt.title('Distribution of Incorrect Predictions by Probability\nfor Random Forest')
plt.show()

# #### Ensembling

# Neural Network Model = model <br>
# Random Forest Model = best_rf <br>
# SGD Model = cal_model (only way to get probabilities for 'huber' loss)

# In[99]:


#### TRAINING DATA for ENSEMBLE ####
nn_probs = np.array([x for sublist in model.predict(x_test1) for x in sublist]) # only predicting on “validation” subset of data
sgd_probs = cal_model.predict_proba(x_test1)[:,1]
rf_probs = best_rf.predict_proba(x_test1)[:,1]

# In[100]:

print(nn_probs.shape)
print(sgd_probs.shape)
print(rf_probs.shape)

# In[101]:

ensemble_train = pd.DataFrame({'neural_net': nn_probs,
                               'sgd_classifier': sgd_probs,
                               'random_forest': rf_probs},
                              index=range(len(nn_probs)))
ensemble_train

# In[102]:

#### TEST DATA for ENSEMBLE ####
nn_probs = np.array([x for sublist in model.predict(x_test2) for x in sublist]) # evaluating on the “holdout” test dataset
sgd_probs = cal_model.predict_proba(x_test2)[:,1]
rf_probs = best_rf.predict_proba(x_test2)[:,1]

# In[103]:

print(nn_probs.shape)
print(sgd_probs.shape)
```

```
print(rf_probs.shape)

# In[104]:

ensemble_test = pd.DataFrame({'neural_net': nn_probs,
                              'sgd_classifier': sgd_probs,
                              'random_forest': rf_probs},
                             index=range(len(nn_probs)))
ensemble_test

# In[105]:

ensemble_model = tf.keras.Sequential([
    tf.keras.layers.Dense(256, input_shape=(3,), activation='relu'),
    tf.keras.layers.Dense(512, activation='relu'),
    tf.keras.layers.Dense(512, activation='tanh'),
    tf.keras.layers.Dense(256, activation='tanh'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# In[106]:

ensemble_model.compile(
    optimizer='adadelta',
    loss='binary_crossentropy',
    metrics=[
        tf.keras.metrics.BinaryAccuracy(name='accuracy'),
        tf.keras.metrics.Precision(name='precision'),
        tf.keras.metrics.Recall(name='recall')
    ]
)

# In[107]:

ensemble_model.summary()

# In[108]:

stops = EarlyStopping(monitor='val_loss', patience=8, min_delta=1e-6)

ensemble_model.fit(ensemble_train, y_test1,
                   validation_data=(ensemble_test, y_test2),
                   epochs=1000,
                   batch_size=10000,
                   callbacks=[stops])

# In[109]:

final_preds = ensemble_model.predict_classes(ensemble_test)
f1_score(y_test2, final_preds)

# In[110]:

font = {'size'   : 13}
plt.rc('font', **font)

cm = confusion_matrix(y_test2, final_preds)
c_disp = ConfusionMatrixDisplay(cm, display_labels=np.asarray([0,1]))
```

```
fig, ax = plt.subplots(figsize=(5,5))
ax.grid(False)
plt.title('Confusion Matrix\nfor Ensemble - Neural Network\n', fontsize=14)
c_disp.plot(cmap=plt.cm.Blues, ax=ax, values_format='')

# In[111]:

final_pred_list = [item for sublist in final_preds.tolist() for item in sublist]
final_plot = pd.DataFrame({'actual':list(y_test2),
                           'predictions':final_pred_list})

final_plot['actual'][0] == final_plot['predictions'][0]
final_plot['comparison'] = np.where(final_plot['actual']==final_plot['predictions'], True, False)
final_plot.head(15)

# In[112]:

final_pred_probs = ensemble_model.predict(ensemble_test)
pd.DataFrame(final_pred_probs[final_plot['comparison']==False]).hist(bins=100, figsize = (12,5))
plt.title('Distribution of Incorrect Predictions by Probability\nfor Ensemble - Final NN')
plt.show()
```

Appendix B – Custom F1 score (weighted values)

Logic below can be used to create a custom F1 metric that applies weights to each input (precision and recall). This will enable tuning model to increase penalty for false positives, given that false positive has higher cost ($100) as compared to false negative (loss = $20). Results from this method are comparable with baseline F1 (50/50 split).

```
# custom F1 v02

import keras.backend as K

def f1_custom_02(y_true, y_pred):
    def recall(y_true, y_pred):
        true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
        possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
        recall = true_positives / (possible_positives + K.epsilon()) # the K.epsilon solves for divide by zero error
        # recall = true_positives / (possible_positives)
        return recall

    def precision(y_true, y_pred):
        true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
        predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
        precision = true_positives / (predicted_positives + K.epsilon())
        return precision

    precision = precision(y_true, y_pred)
    recall = recall(y_true, y_pred)

    # add weights to change balance of F1 score
    # (precision) false positive costs $100
    # (recall) false negative costs $20
    # we want to penalize false positive more than false negative
    # custom_f1 = custom_score(recall_weight=0.17, precision_weight=0.83)
    precision_weight = 0.83
    recall_weight = 0.17
```

```
    # precision = precision*precision_weight
    # recall = recall*recall_weight

    return (recall_weight*recall + precision*precision_weight)

# model.compile(loss='binary_crossentropy',
#           optimizer= "adam",
#           metrics=[f1])

nn_model.compile(optimizer = 'adam',
                 loss = 'binary_crossentropy',
                 metrics = [f1_custom_02,
                            tf.keras.metrics.BinaryAccuracy(name = 'accuracy'),
                            tf.keras.metrics.Precision(name = 'precision'),
                            tf.keras.metrics.Recall(name = 'recall')
                            ]
                 )

stops = EarlyStopping(monitor='val_loss', patience=3, min_delta=1e-5)

model_fit_nn = nn_model.fit(x_train_nn, y_train_nn,
                            validation_data=(x_val_nn, y_val_nn),
                            epochs=1000,
                            batch_size=10000,
                            callbacks=[stops])
```