

# Binary\_Classification\_Report (1)

April 11, 2021

## 0.1 Task 1 Relevance Modelling: Predict the relevance of documents given search interaction features.

Class: CS987

Group: AE

- Andrea Oteo: 202059017
- Ane Etxeandia Erauskin: 202077594
- Crawford Middleton: 202082754
- Daragh McCarthy: 202075755
- Elijah Reid: 202093659

## 1 Overview and Abstract

Briefly describe what you did

- We began by exploring the data to see what values seemed likely to have an effect on whether the document was classified as relevant or not. Following this, we dropped the variables that we deemed as irrelevant and then used one-hot encoding to transform the categorical variables into numerical ones so they could be better understood by our algorithms. After this we split the data into training, validation and test sets using `sci-learn train_test_split` before scaling the data to give it a mean of 0 and standard deviation of 1.
- We fed this into a standard machine learning model (random forest classifier) but due the highly skewed nature of the data (relevant documents make up only ~6% of the entire dataset) the performance was not great.
- To alleviate this we created a three-layer neural network as a baseline and experimented with various techniques to overcome the imbalance. This included oversampling, under-sampling, ensemble methods, SMOTE, setting the initial bias and class weights to see which would deliver the best performance.
- We also performed data augmentation on the `cpvs` column to both increase the number of features and rows. We exploded the list to increase length, and then shortened the code to the first two numbers to represent 'division' before one-hot encoding it.
- In the end, the method that gave the best performance was adjusting the class weights to better fit the data, so we decided to move forward using the class weights of the three layer model for our next two models.

- We created a neural network with several more layers than the initial baseline to see if a deeper network would perform better than a shallow one. This turned out to be true although the performance was only mildly better.
- Finally we created a wide and deep network, splitting the inputs into a relevant and irrelevant training set so the network could better learn how to distinguish between the two and hopefully improve it's classification performance. Ultimately the model's performance was unsatisfactory as it quickly overfit the training data and ended up predicting every document as being irrelevant.

## 2 Method

### 2.0.1 Import Modules

```
[ ]: # Include your packages/imports here.
!pip install -q pyyaml h5py

from google.colab import drive

import tensorflow as tf
from tensorflow import keras

import os
import tempfile

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

import sklearn
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, f1_score, ↵
    ↪precision_score, recall_score, roc_curve, roc_auc_score, classification_report
from sklearn.model_selection import train_test_split, StratifiedKFold, ↵
    ↪GridSearchCV, RandomizedSearchCV, cross_val_score, cross_validate
from sklearn.preprocessing import StandardScaler, MinMaxScaler, LabelEncoder

from keras.models import Sequential
from keras.layers import Dense, Flatten, Activation, Dropout
from keras.callbacks import Callback
from keras.utils import normalize, to_categorical
from keras.wrappers.scikit_learn import KerasClassifier
```

```

from ast import literal_eval
from functools import partial

mpl.rcParams['figure.figsize'] = (12, 10)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']

```

## 2.1 Describe the data processing, feature extraction, etc. performed (and why it was performed)

After the data was loaded in, we began by dropping columns that contained too many unique values. This was because our approach was to one-hot encode categorical variables and if we had encoded these variables the model would have ended up overfitting on the training set due to the large number of variables.

While we initially considered dropping the CPVs column for the above reasons, we instead shortened the code to it's first two digits (this refers to the code's division category) to reduce the number of unique variations in the dataset and then encoded this column as well.

## 2.2 Describe the baseline model to be used (and why it was selected)

The standard machine learning base line algorithm we have selected is the random forest model. Random forest algorithm is an ensemble of decision trees, which is trained using the technique of bagging.

During training the model is focused on selecting the best features instead of instances. The algorithm selects the best features from a random subset of features.

The model is appropriate for several reasons: - the algorithm has high bias/low variance. This is ideal for the dataset as it has a large number of observations compared to the number of features. - it allows us to analyse feature importance by describing the most applicable features to impact a user assessing a document as relevant or not.

## 2.3 Describe each neural model configuration / setup that will be used.

The model's we have decided to include are the following: - Three Layer MLP model - Dense Deep Model (ultimately 6 layers were selected) - A Wide and Deep Model as we believed this may have helped to reduce the effect of the bias towards irrelevant documents featured in the data (more on this below)

Other models were trialled, including a Deep Belief Network and a Natural Language Model but ultimately, they performed worse both in terms of their f1 score and the time needed to train them.

## 2.4 Describe the training schedule and approach that you undertook.

While training we used several methods to improve the models performance and limit it from overfitting.

We created a custom dense layer to use, which made use of: - the ELU activation function (This is slower to compute than the standard RELU function, but it makes up for it by tending to converge

faster during training) - the He initialiser (this initialisation method is optimised for ReLU variations) - l2 regularisation (this helps constrain the neural network's connection weights and limits overfitting)

When creating our models, we added both batch normalisation and drop out layers: - Batch normalisation works with the He initialiser and ELU to ensure that the vanishing/exploding gradient problem does not occur by normalising and zero-centring each input and then scaling and shifting the result based on this. - The drop out layer works by giving each neuron a random chance to be dropped out at each training step, helping to prevent overreliance on specific connections and decreasing the error rate in validation.

We decided to use the Nadam optimiser because it is an adaptive optimiser- it combines the Adam optimiser (which is already able to keep track of an exponentially decaying average of past gradient like momentum optimisation, and can keep track of past squared gradients like RMSProp) with the Nesterov trick, allowing it to run slightly faster than the standard Adam optimiser.

Finally, the Learning Rate Scheduler we used was 1cycle scheduling. Unlike most other approaches it increases the learning rate linearly up to halfway through training and then decreases it in the second half, dropping down several orders of magnitude in the last few epochs. We selected it as it has been shown to produce similar validation accuracy results to other methods, but in a far shorter time span.

## 2.5 Define the functions that will help you to perform the training schedule

```
[ ]: # Create Dense layer with built in selu function and appropriate regularisation
RegularizedDense = partial(keras.layers.Dense,
                           activation="elu",
                           kernel_initializer="he_normal",
                           kernel_regularizer=keras.regularizers.l1_l2(0.01))

WDRegularizedDense = partial(keras.layers.Dense,
                              activation="selu",
                              kernel_initializer="lecun_normal",
                              kernel_regularizer=keras.regularizers.l1_l2(0.01))

# Sets up all the metrics that allow for the f1_score to be calculated
METRICS = [
    keras.metrics.TruePositives(name='tp'),
    keras.metrics.FalsePositives(name='fp'),
    keras.metrics.TrueNegatives(name='tn'),
    keras.metrics.FalseNegatives(name='fn'),
    keras.metrics.BinaryAccuracy(name='accuracy'),
    keras.metrics.Precision(name='precision'),
    keras.metrics.Recall(name='recall'),
    keras.metrics.AUC(name='auc'),
]

# Implement early stopping to help limit overfitting
early_stopping = tf.keras.callbacks.EarlyStopping(
```

```

monitor='val_precision',
verbose=1,
patience=10,
mode='max',
restore_best_weights=True)

# Custom One Cycle Learning Scheduler for use with models
class OneCycleScheduler(keras.callbacks.Callback):
    def __init__(self, iterations, max_rate, start_rate=None,
                  last_iterations=None, last_rate=None):
        self.iterations = iterations
        self.max_rate = max_rate
        self.start_rate = start_rate or max_rate / 10
        self.last_iterations = last_iterations or iterations // 10 + 1
        self.half_iteration = (iterations - self.last_iterations) // 2
        self.last_rate = last_rate or self.start_rate / 1000
        self.iteration = 0
    def _interpolate(self, iter1, iter2, rate1, rate2):
        return ((rate2 - rate1) * (self.iteration - iter1)
                / (iter2 - iter1) + rate1)
    def on_batch_begin(self, batch, logs):
        if self.iteration < self.half_iteration:
            rate = self._interpolate(0, self.half_iteration, self.start_rate,
→self.max_rate)
        elif self.iteration < 2 * self.half_iteration:
            rate = self._interpolate(self.half_iteration, 2 * self.
→half_iteration,
                                   self.max_rate, self.start_rate)
        else:
            rate = self._interpolate(2 * self.half_iteration, self.iterations,
                                   self.start_rate, self.last_rate)
        self.iteration += 1
        K.set_value(self.model.optimizer.lr, rate)

```

## 2.6 Describe any other things that you did or tried in order to improve performance

As the dataset was heavily skewed towards results that were irrelevant (there were only 1962 values for 1 out of 33000 total records - only 5.95% of the total)

To try and alleviate this we used several methods and compared their performance. These are as follows: - We adjusted the biases of the model to try and prevent the model from overfitting on the irrelevant documents (0) and to focus more on the relevant documents (1) to compensate for the vastly different amounts of each. This only applies to the output step however which is less than ideal.

- This did have some effect on the number of false positives, however it led to more false negatives, so we instead decided to adjust the class weights directly. This causes the model to “pay more attention” to examples from an under-represented class” during the training

process.

- As a final option we used SMOTE to resample the dataset and oversample the minority class. This means that no adjustments need to be made to the actual model and should improve its ability to recognise what a relevant document looks like.

After trying all 3 methods with our three-layer neural network, both the re-weighted and resampled models performed significantly better than with the models trained on the original data so we continued to use these methods for the dense deep network.

### 2.6.1 Set-up functions

```
[ ]: # Add your functions for training here

# one hot encodes categorical variables to allow them to be used in neural_
→network training
def encode_and_bind(original_dataframe, feature_to_encode):
    dummies = pd.get_dummies(original_dataframe[feature_to_encode])
    res = pd.concat([original_dataframe, dummies], axis=1)
    res = res.drop([feature_to_encode], axis=1)
    return(res)

# Function to display mean and standard deviation of scores for standard machine_
→learning model
def display_scores(scores):
    print("\nScores:", scores)
    print("\nMean:", scores.mean())
    print("\nStandard Deviation:", scores.std())

# Function to create confusion matrices with various statistics and scores
def plot_cm(labels, predictions, p=0.5):
    cm = confusion_matrix(labels, predictions > p)
    plt.figure(figsize=(5,5))
    sns.heatmap(cm, annot=True, fmt="d")
    plt.title('Confusion matrix @{: .2f}'.format(p))
    plt.ylabel('Actual label')
    plt.xlabel('Predicted label')
    print('f1 Score', (cm[1][1]/(cm[1][1] + (0.5*(cm[0][1] + cm[1][0])))))
    print()

    print('Irrelevant Documents Detected (True Negatives): ', cm[0][0])
    print('Irrelevant Documents Incorrectly Detected (False Positives): ',
→cm[0][1])
    print('Relevant Documents Missed (False Negatives): ', cm[1][0])
    print('Relevant Documents Detected (True Positives): ', cm[1][1])
    print('Total Relevant Documents: ', np.sum(cm[1]))

# Function to compare loss performance of two models
```

```

def plot_loss(history, label, n):
    # Use a log scale on y-axis to show the wide range of values.
    plt.semilogy(history.epoch, history.history['loss'],
                  color=colors[n], label='Train ' + label)
    plt.semilogy(history.epoch, history.history['val_loss'],
                  color=colors[n], label='Val ' + label,
                  linestyle="--")
    plt.xlabel('Epoch')
    plt.ylabel('Loss')

# Function to produce plots of metrics for model training
def plot_metrics(history):
    metrics = ['loss', 'auc', 'precision', 'recall']
    for n, metric in enumerate(metrics):
        name = metric.replace("_", " ").capitalize()
        plt.subplot(2,2,n+1)
        plt.plot(history.epoch, history.history[metric], color=colors[0],
        →label='Train')
        plt.plot(history.epoch, history.history['val_'+metric],
                  color=colors[0], linestyle="--", label='Val')
        plt.xlabel('Epoch')
        plt.ylabel(name)
        if metric == 'loss':
            plt.ylim([0, plt.ylim()[1]])
        elif metric == 'auc':
            plt.ylim([0,1])
        else:
            plt.ylim([0,1])

        plt.legend()

# Function for producing an ROC plot to determine precision and recall of model
def plot_roc(name, labels, predictions, **kwargs):
    fp, tp, _ = sklearn.metrics.roc_curve(labels, predictions)

    plt.plot(100*fp, 100*tp, label=name, linewidth=2, **kwargs)
    plt.xlabel('False positives [%]')
    plt.ylabel('True positives [%]')
    plt.xlim([-0.5,100])
    plt.ylim([0,100.5])
    plt.grid(True)
    ax = plt.gca()
    ax.set_aspect('equal')

```

## 2.6.2 Models

**Final Random Forest Model (Baseline)** <https://colab.research.google.com/drive/1boa4ZFjmB7C0jcKh6fugi14xTP#scrollTo=5hh8XILyN36C&line=7&uniqifier=1>

**Three Layer Model Function** <https://colab.research.google.com/drive/1boa4ZFjmB7C0jcKh6fugi14CXfhQ-xTP#scrollTo=A88mkimt0IDN&line=11&uniqifier=1>

**Dense Deep Model Function** [https://colab.research.google.com/drive/1boa4ZFjmB7C0jcKh6fugi14CXfhQ-xTP#scrollTo=\\_yMzpAIBxXHB&line=6&uniqifier=1](https://colab.research.google.com/drive/1boa4ZFjmB7C0jcKh6fugi14CXfhQ-xTP#scrollTo=_yMzpAIBxXHB&line=6&uniqifier=1)

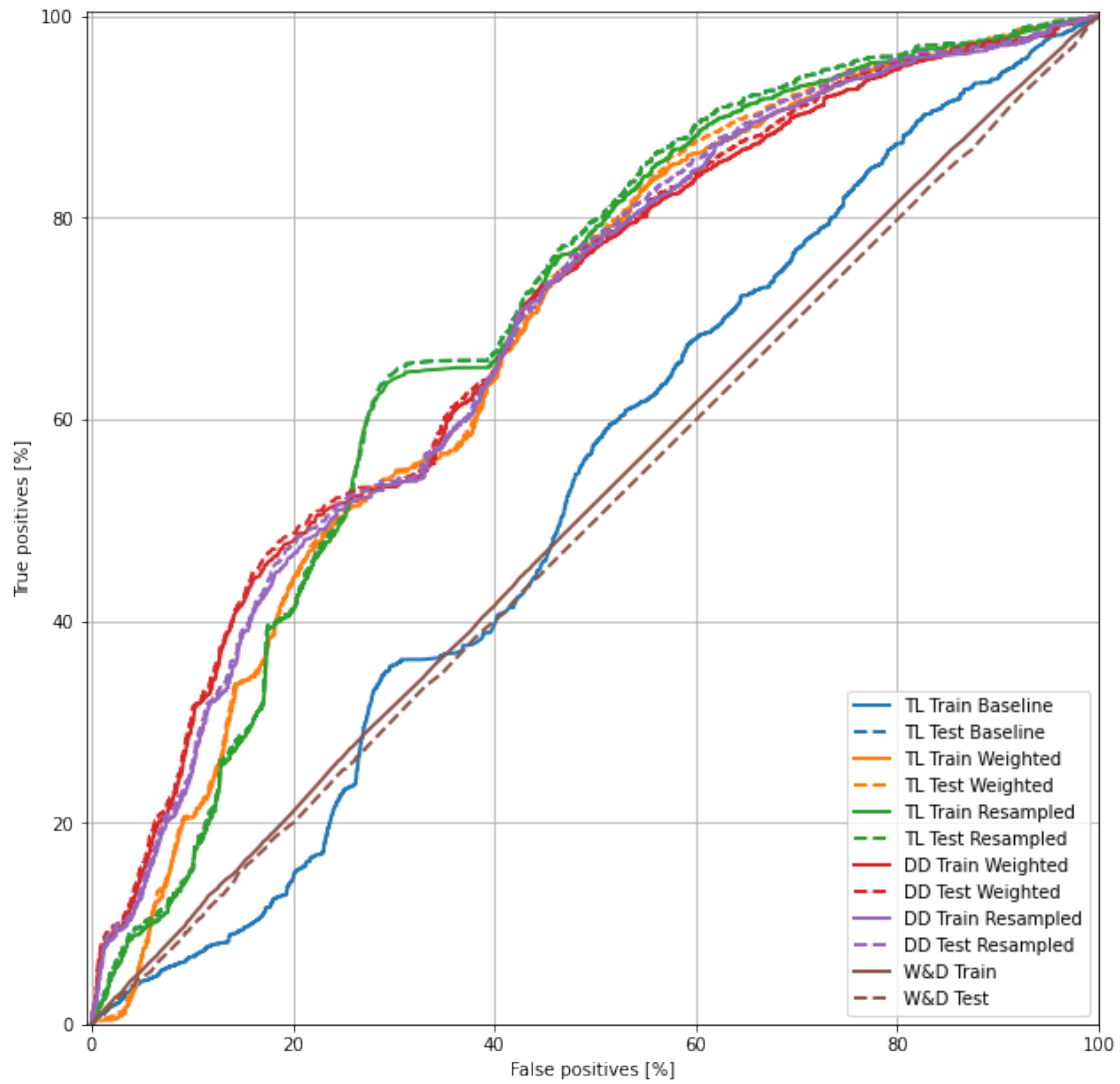
**Wide and Deep Model Class** <https://colab.research.google.com/drive/1boa4ZFjmB7C0jcKh6fugi14CXfhQ-xTP#scrollTo=Nky1aIvoWBwt&line=6&uniqifier=1>

## 3 Results and Discussion

Model Scores on validation data, loss has been set to 0 for random forests due to the difference of their architectures

The Wide and Deep Model also score highly but this is due to it's f1 score only reflecting it's ability to predict if a document is irrelevant, which it predicted every document as





Graph of each model's train and test performance, comparing percentage of true positives to percentage of false positives

	Model	Loss	Accuracy	Precision	Recall	F1 Score
0	Unweighted Random Forest	0.000000	0.929103	1.000000	0.000738	0.001475
1	Weighted Random Forest	0.000000	0.692394	0.174355	0.892949	0.291745
2	Adjusted Bias Three Layer Model	8.002959	0.929050	0.000000	0.000000	0.000000
3	Adjusted Weights Three Layer Model	6.251603	0.921429	0.277863	0.067183	0.108205
4	Resampled Three Layer Model	4.618025	0.555890	0.108872	0.732004	0.189552
5	Dense Deep Model	14.210053	0.095988	0.071944	0.986711	0.134109
6	Resampled Dense Deep Layer Model	6.606988	0.473888	0.100317	0.805094	0.178405
7	Wide and Deep Model	0.257029	0.929050	0.929050	0.929050	0.929050

The model that performed the best was the resampled dense deep network, with an f1 score on the test set of (0.10633). However this is only mildly better than the three layer models trained with adjusted class weights and resampled data which score (0.09920) and (0.09267) respectively.

Contrary to our expectations the wide and deep model actually performed the worst as even with a limited epoch run it would very quickly overfit the data and predict all documents as being irrelevant, which scored very well on the skewed training set but poorly on the test set as a result.

Unfortunately even with the methods we've used to try and improve the model's ability to recognise relevant documents, even our best performing model was only able to achieve an F1 score of (0.10633).

	Model	Neurons per Layer	Number of Layers	Kernel Initializer	Activation Function	Normalization	Regularisation	Optimizer	Learning Rate Schedule
0	Three Layer Model	30	3	He Initialization	ELU	Batch Normalisation	l1 and l2 regularisation with Dropout if needed	Nadam	1cycle
1	Dense Deep Model	30	5	He Initialization	ELU	Batch Normalisation	l1 and l2 regularisation with Dropout if needed	Nadam	1cycle
2	Wide and Deep Model	30	5 (2 inputs, 2 dense and output)	LeCun Initialization	SELU	None (Self-Normalisation)	l1 and l2 regularisation	Nadam	1cycle

## 4 Summary and Recommendations

Based off our results we would recommend that the company should go with our three layer model, as it was the most able to generalise and predict accurately on the test data. The deep dense network also shows promise, and with more in-depth tuning it seems likely that it would be able to outperform the three layer model.

However it still has a lacklustre performance compared to models used for skewed datasets (for instance in credit card fraud detection) and so we would recommend that the company should gather more data points for relevant documents and use these for training their model. Using an embedding layer to include more of the categorical data, and the associations between them would also probably yield better performances.

Alternatively they could train a Generative Adversarial network on the dataset, which would allow them to generate new datapoints and massively improve the performance on the dataset. A Natural Language Model might also be able to perform better on the dataset by reading in the data in text form and converting the cvps codes into their text descriptions.

## 5 References

- Aurélien Géron (September 2019) *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition edn.*, US: O'Reilly Media, Inc.
- TensorFlow (2020) *Classification on imbalanced data*, Available at: [https://www.tensorflow.org/tutorials/structured\\_data/imbalanced\\_data](https://www.tensorflow.org/tutorials/structured_data/imbalanced_data)
- Andrej Karpathy (2019) *A Recipe for Training Neural Networks*, Available at: <http://karpathy.github.io/2019/04/25/recipe/#2-set-up-the-end-to-end-training-evaluation-skeleton-get-dumb-baselines>
- Leslie N. Smith (2018) A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay
- Nitish Srivastava et al (2014) ‘Dropout: A Simple Way to Prevent Neural Networks from Overfitting’, *Journal of Machine Learning Research*, 15(), pp. 1929-1958.

## 6 Code

### 6.1 Feature Processing

#### 6.1.1 Load in dataset from Github

```
[ ]: train_url = "https://raw.githubusercontent.com/Wizzzzzzard/Wizzzzzzard/main/
      ↳CS987/Binary%20Data/train.csv"
test_url = "https://raw.githubusercontent.com/Wizzzzzzard/Wizzzzzzard/main/CS987/
      ↳Binary%20Data/test.csv"
```

```
[ ]: train = pd.read_csv(train_url)
test = pd.read_csv(test_url)
```

```
/usr/local/lib/python3.7/dist-packages/IPython/core/interactiveshell.py:2718:
DtypeWarning: Columns (16) have mixed types.Specify dtype option on import or
set low_memory=False.
interactivity=interactivity, compiler=compiler, result=result)
```

#### 6.1.2 Data processing and exploration

```
[ ]: train.head()
```

```
[ ]:
      user  ... #cpv45
0    8438057  ...     1
1    8438876  ...     2
2   922102585  ...     2
3  2105483652  ...     2
```

```
4      8438876 ...      1
```

```
[5 rows x 19 columns]
```

```
[ ]: test.head()
```

```
[ ]:      user      session ... #cpv45 Id
0  2096178939  3C5FDDE0DBC2E5E812A8DFFAB3491DAA ...      1  0
1  2096178939  B9D21C26929EF384ABBB6B544FB38858 ...      1  1
2  2096178939  B9D21C26929EF384ABBB6B544FB38858 ...      1  2
3  2096178939  B9D21C26929EF384ABBB6B544FB38858 ...      2  3
4  2096178939  71182AF6B9BCB557CFA9402F6CD97361 ...      1  4
```

```
[5 rows x 19 columns]
```

```
[ ]: # Examine the class label imbalance

neg, pos = np.bincount(train['psrel'])
total = neg + pos
print('Examples:\n      Total: {} \n      Ones: {} ({:.2f}% of total)\n'.format(
      total, pos, 100 * pos / total))
```

Examples:

```
Total: 33000
```

```
Ones: 1962 (5.95% of total)
```

### 6.1.3 Clean, split and normalize the data

The raw data has a few issues. First the Time columns as well as the user, session and query columns are too variable to use directly. Drop the Time and other columns (since it's not clear how they would relate to the relevancy in this context) and drop the #cpvs45 column to remove the colinearity between it and cpvs

```
[ ]: cleaned_df = train.copy()

# drop variables
cleaned_df = cleaned_df.drop(["user", "session", "query", "timestamp", "#cpv45",
    → "day", "hour", "month"], axis=1)
test = test.drop(["user", "session", "query", "timestamp", "#cpv45", "day",
    → "hour", "month", "Id"], axis=1)
```

```
[ ]: # augment cpvs - train - lengthened and widened
cleaned_df['cpvs'] = cleaned_df['cpvs'].apply(literal_eval) #convert to list type

# explode cpvs list to lengthen dataset
cleaned_df = cleaned_df.explode('cpvs')
```

```
# shorten cpvs column to first two numbers which represent division category
cleaned_df['division'] = cleaned_df['cpvs'].astype(str).str[:2]
cleaned_df = cleaned_df.drop("cpvs", axis=1) #drop new column

# augment cpvs - test - only widened - keep 5000 rows
test['cpvs'] = test['cpvs'].apply(literal_eval) #convert to list type
test["cpv_new"] = test["cpvs"].str[0] #copy first entry from each list
test = test.drop("cpvs", axis=1) #drop original column

test["division"] = test['cpv_new'].astype(str).str[:2]
test = test.drop("cpv_new", axis=1) #drop new column column
```

```
[ ]: # One-hot encode the categorical variables that seem like they may have an
      ↪effect on the relevancy of the document found

features_to_encode = ['search', 'source', 'type', 'nature', 'division']

for feature in features_to_encode:
    cleaned_df = encode_and_bind(cleaned_df, feature)

for feature in features_to_encode:
    test = encode_and_bind(test, feature)
```

## 6.2 Training and Validating etc.

- Show your working here – where you report all your training and validation, etc. that you performed in order to get the results.
- Note that it is important that you results can be replicated. All code to reproduce the final predictions must be included, along with any code that justifies your choices.

Split the dataset into train, validation, and test sets.

- The validation set is used during the model fitting to evaluate the loss and any metrics, however the model is not fit with this data.
- The test set is completely unused during the training phase and is only used at the end to evaluate how well the model generalizes to new data.

This is especially important in our case due to the imbalanced dataset as [overfitting](#) is a significant concern from the lack of training data.

```
[ ]: # Use sklearn train_test_split to split and shuffle our dataset.
train_df, test_df = train_test_split(cleaned_df, test_size=0.2)
train_df, val_df = train_test_split(train_df, test_size=0.2)

train_wide_and_deep_df, test_wide_and_deep_df = train_test_split(cleaned_df,
    ↪test_size=0.2)
train_wide_and_deep_df, val_wide_and_deep_df =
    ↪train_test_split(train_wide_and_deep_df, test_size=0.2)
```

```

# Form np arrays of labels and features.
train_labels = np.array(train_df.pop('psrel'))
bool_train_labels = train_labels != 0
val_labels = np.array(val_df.pop('psrel'))
test_labels = np.array(test_df.pop('psrel'))

train_features = np.array(train_df)
val_features = np.array(val_df)
test_features = np.array(test_df)

```

```

[ ]: # wide and deep model feature selection
# -----

## Train
# search
print("advanced: ", train_wide_and_deep_df.columns.get_loc("advanced"))
print("saved: ", train_wide_and_deep_df.columns.get_loc("saved"))
print("dropdown: ", train_wide_and_deep_df.columns.get_loc("dropdown"))
print("quick: ", train_wide_and_deep_df.columns.get_loc("quick"))
# source
print("Contracts Finder: ", train_wide_and_deep_df.columns.get_loc("Contracts_
    ↳Finder"))
print("Contrax Weekly: ", train_wide_and_deep_df.columns.get_loc("Contrax_
    ↳Weekly"))
print("Defence Contracts International: ", train_wide_and_deep_df.columns.
    ↳get_loc("Defence Contracts International"))
print("EBS: ", train_wide_and_deep_df.columns.get_loc("EBS"))
print("Exporting Opportunity: ", train_wide_and_deep_df.columns.
    ↳get_loc("Exporting Opportunity"))
print("FedCon: ", train_wide_and_deep_df.columns.get_loc("FedCon"))
print("Glenigan: ", train_wide_and_deep_df.columns.get_loc("Glenigan"))
print("Intercon: ", train_wide_and_deep_df.columns.get_loc("Intercon"))
print("MoD Contracts Bulletin: ", train_wide_and_deep_df.columns.get_loc("MoD_
    ↳Contracts Bulletin"))
print("PCS: ", train_wide_and_deep_df.columns.get_loc("PCS"))
print("Project: ", train_wide_and_deep_df.columns.get_loc("Project"))
print("Tracker: ", train_wide_and_deep_df.columns.get_loc("Tracker"))

# type
print("notice: ", train_wide_and_deep_df.columns.get_loc("notice"))
print("award: ", train_wide_and_deep_df.columns.get_loc("award"))
print("adden: ", train_wide_and_deep_df.columns.get_loc("adden"))
print("tenis: ", train_wide_and_deep_df.columns.get_loc("tenis"))

# nature
print("services: ", train_wide_and_deep_df.columns.get_loc("services"))

```

```

print("supplies: ", train_wide_and_deep_df.columns.get_loc("supplies"))
print("works: ", train_wide_and_deep_df.columns.get_loc("works"))

# psrel
print("psrel: ", train_wide_and_deep_df.columns.get_loc("psrel"))

## Test
# search
print("advanced: ", test.columns.get_loc("advanced"))
print("saved: ", test.columns.get_loc("saved"))
print("dropdown: ", test.columns.get_loc("dropdown"))
print("quick: ", test.columns.get_loc("quick"))
# source
print("Contracts Finder: ", test.columns.get_loc("Contracts Finder"))
print("Contrax Weekly: ", test.columns.get_loc("Contrax Weekly"))
print("Defence Contracts International: ", test.columns.get_loc("Defence_
↳Contracts International"))
print("EBS: ", test.columns.get_loc("EBS"))
print("Exporting Opportunity: ", test.columns.get_loc("Exporting Opportunity"))
print("FedCon: ", test.columns.get_loc("FedCon"))
print("Glenigan: ", test.columns.get_loc("Glenigan"))
print("Intercon: ", test.columns.get_loc("Intercon"))
print("MoD Contracts Bulletin: ", test.columns.get_loc("MoD Contracts Bulletin"))
print("PCS: ", test.columns.get_loc("PCS"))
print("Project: ", test.columns.get_loc("Project"))
print("Tracker: ", test.columns.get_loc("Tracker"))

# type
print("notice: ", test.columns.get_loc("notice"))
print("award: ", test.columns.get_loc("award"))
print("adden: ", test.columns.get_loc("adden"))
print("tenis: ", test.columns.get_loc("tenis"))

# nature
print("services: ", test.columns.get_loc("services"))
print("supplies: ", test.columns.get_loc("supplies"))
print("works: ", test.columns.get_loc("works"))

```

```

advanced: 6
saved: 9
dropdown: 7
quick: 8
Contracts Finder: 10
Contrax Weekly: 11
Defence Contracts International: 12
EBS: 13
Exporting Opportunity: 14

```

FedCon: 15  
 Glenigan: 16  
 Intercon: 17  
 MoD Contracts Bulletin: 18  
 PCS: 19  
 Project: 20  
 Tracker: 21  
 notice: 24  
 award: 23  
 adden: 22  
 tennis: 25  
 services: 26  
 supplies: 27  
 works: 28  
 psrel: 5  
 advanced: 5  
 saved: 8  
 dropdown: 6  
 quick: 7  
 Contracts Finder: 9  
 Contrax Weekly: 10  
 Defence Contracts International: 11  
 EBS: 12  
 Exporting Opportunity: 13  
 FedCon: 14  
 Glenigan: 15  
 Intercon: 16  
 MoD Contracts Bulletin: 17  
 PCS: 18  
 Project: 19  
 Tracker: 20  
 notice: 23  
 award: 22  
 adden: 21  
 tennis: 24  
 services: 25  
 supplies: 26  
 works: 27

```

[ ]: wide_train_df = train_wide_and_deep_df.iloc[:,
    ↳ [5,6,9,7,8,10,11,12,13,14,15,16,17,18,19,20,21,24,23,22,25,26,27,28]]
    deep_train_df = train_wide_and_deep_df

    wide_val_df = val_wide_and_deep_df.iloc[:,
    ↳ [5,6,9,7,8,10,11,12,13,14,15,16,17,18,19,20,21,24,23,22,25,26,27,28]]
    deep_val_df = val_wide_and_deep_df
  
```



```

wide_test_df = test_wide_and_deep_df.iloc[:,  

    ↳[5,6,9,7,8,10,11,12,13,14,15,16,17,18,19,20,21,24,23,22,25,26,27,28]]
deep_test_df = test_wide_and_deep_df

wide_test = test.iloc[:,  

    ↳[5,8,6,7,9,10,11,12,13,14,15,16,17,18,19,20,23,22,21,24,26,26,27]]
deep_test = test

```

```

[ ]: # Form np arrays of labels and features.
wide_train_labels = np.array(wide_train_df.pop('psrel'))
wide_bool_train_labels = wide_train_labels != 0
wide_val_labels = np.array(wide_val_df.pop('psrel'))
wide_test_labels = np.array(wide_test_df.pop('psrel'))

wide_train_features = np.array(wide_train_df)
wide_val_features = np.array(wide_val_df)
wide_test_features = np.array(wide_test_df)

#wide_test = np.array(wide_test)

```

```

[ ]: # Form np arrays of labels and features.
deep_train_labels = np.array(deep_train_df.pop('psrel'))
deep_bool_train_labels = deep_train_labels != 0
deep_val_labels = np.array(deep_val_df.pop('psrel'))
deep_test_labels = np.array(deep_test_df.pop('psrel'))

deep_train_features = np.array(deep_train_df)
deep_val_features = np.array(deep_val_df)
deep_test_features = np.array(deep_test_df)

#deep_test = np.array(deep_test)

```

Normalize the input features using the sklearn StandardScaler - this will set the mean to 0 and standard deviation to 1.

Only the StandardScaler is fit using the train\_features to ensure the model is not gaining any information about the validation or test sets.

```

[ ]: scaler = StandardScaler()

train_df, test_df = train_test_split(cleaned_df, test_size=0.2)

X = cleaned_df.drop(labels="psrel", axis=1)
y = cleaned_df['psrel']

X_train, X_test, y_train, y_test = train_test_split(X, y)

train_features = scaler.fit_transform(train_features)

```

```

val_features = scaler.transform(val_features)
test_features = scaler.transform(test_features) # This is to be used to compare
→the predictions after training has occurred
test = scaler.transform(test) # this is the originally imported test that will
→be used for final predictions

train_features = np.clip(train_features, -5, 5)
val_features = np.clip(val_features, -5, 5)
test_features = np.clip(test_features, -5, 5)
test = np.clip(test, -5, 5)

print('Training labels shape:', train_labels.shape)
print('Validation labels shape:', val_labels.shape)
print('Test labels shape:', test_labels.shape)

print('Training features shape:', train_features.shape)
print('Validation features shape:', val_features.shape)
print('Test features shape:', test_features.shape)

print('Test shape', test.shape)

```

```

Training labels shape: (122179,)
Validation labels shape: (30545,)
Test labels shape: (38182,)
Training features shape: (122179, 74)
Validation features shape: (30545, 74)
Test features shape: (38182, 74)
Test shape (5000, 74)

```

```
[ ]: wide_test.shape
```

```
[ ]: (5000, 23)
```

```

[ ]: wide_train_features = scaler.fit_transform(wide_train_features)

wide_val_features = scaler.transform(wide_val_features)
wide_test_features = scaler.transform(wide_test_features) # This is to be used
→to compare the predictions after training has occurred
wide_test = scaler.transform(wide_test) # this is the originally imported test
→that will be used for final predictions

wide_train_features = np.clip(wide_train_features, -5, 5)
wide_val_features = np.clip(wide_val_features, -5, 5)
wide_test_features = np.clip(wide_test_features, -5, 5)
wide_test = np.clip(wide_test, -5, 5)

```

```

print('Wide Training labels shape:', wide_train_labels.shape)
print('Wide Validation labels shape:', wide_val_labels.shape)
print('Wide Test labels shape:', wide_test_labels.shape)

print('Wide Training features shape:', wide_train_features.shape)
print('Wide Validation features shape:', wide_val_features.shape)
print('Wide Test features shape:', wide_test_features.shape)

print('Wide Test shape', wide_test.shape)

```

```

Wide Training labels shape: (122179,)
Wide Validation labels shape: (30545,)
Wide Test labels shape: (38182,)
Wide Training features shape: (122179, 23)
Wide Validation features shape: (30545, 23)
Wide Test features shape: (38182, 23)
Wide Test shape (5000, 23)

```

```

[ ]: deep_train_features = scaler.fit_transform(deep_train_features)

deep_val_features = scaler.transform(deep_val_features)
deep_test_features = scaler.transform(deep_test_features) # This is to be used
    →to compare the predictions after training has occurred
deep_test = scaler.transform(deep_test) # this is the originally imported test
    →that will be used for final predictions

deep_train_features = np.clip(deep_train_features, -5, 5)
deep_val_features = np.clip(deep_val_features, -5, 5)
deep_test_features = np.clip(deep_test_features, -5, 5)
deep_test = np.clip(deep_test, -5, 5)

print('Deep Training labels shape:', deep_train_labels.shape)
print('Deep Validation labels shape:', deep_val_labels.shape)
print('Deep Test labels shape:', deep_test_labels.shape)

print('Deep Training features shape:', deep_train_features.shape)
print('Deep Validation features shape:', deep_val_features.shape)
print('Deep Test features shape:', deep_test_features.shape)

print('Deep Test shape', deep_test.shape)

```

```

Deep Training labels shape: (122179,)
Deep Validation labels shape: (30545,)
Deep Test labels shape: (38182,)
Deep Training features shape: (122179, 74)

```

```
Deep Validation features shape: (30545, 74)
Deep Test features shape: (38182, 74)
Deep Test shape (5000, 74)
```

### ###Standard Machine Learning Model Baseline - Random Forest (Decision Tree Ensemble)

```
[ ]: # perform a grid search to determine optimal hyperparameters for this model

"""param_grid = {
    'criterion': ["gini", "entropy"],
    'max_features': ["auto", "sqrt", "log2"],
    'max_leaf_nodes': [1, 10, 100],
    'min_samples_leaf': [1, 10, 100],
    'n_estimators': [200, 250, 300]
}

rnd_clf = RandomForestClassifier()

grid_search = GridSearchCV(estimator = rnd_clf, param_grid = param_grid,
                           cv = 5, n_jobs = -1)
grid_search.fit(train_features, train_labels)
grid_search.best_params_"""

[ ]: 'param_grid = {\n    \'criterion\': ["gini", "entropy"],\n    \'max_features\':
["auto", "sqrt", "log2"],\n    \'max_leaf_nodes\': [1, 10, 100],\n
\'min_samples_leaf\': [1, 10, 100],\n    \'n_estimators\': [200, 250,
300]\n}\n\nrnd_clf = RandomForestClassifier()\n\ngrid_search =
GridSearchCV(estimator = rnd_clf, param_grid = param_grid, \n
cv = 5, n_jobs = -1)\ngrid_search.fit(train_features,
train_labels)\ngrid_search.best_params_'
```

```
[ ]: # Standard Machine Learning Model
rnd_clf = RandomForestClassifier(criterion = "gini",
                                n_estimators=200,
                                max_leaf_nodes=100,
                                min_samples_leaf=1,
                                max_features="sqrt",
                                n_jobs=-1)
```

```
[ ]: rnd_clf.fit(train_features, train_labels)

pred_labels = rnd_clf.predict(test_features)

scores = cross_validate(rnd_clf, train_features, train_labels,
                        scoring=('accuracy'),cv=10)

unwt_rnd = {'Model': 'Unweighted Random Forest',
            'Loss': 0,
```

```

'Accuracy': accuracy_score(test_labels, pred_labels),
'Precision': precision_score(test_labels, pred_labels),
'Recall': recall_score(test_labels, pred_labels),
'F1 Score': f1_score(test_labels, pred_labels)}

```

unwt\_rnd

```

[ ]: {'Accuracy': 0.9302550940233618,
      'F1 Score': 0.010405053883314752,
      'Loss': 0,
      'Model': 'Unweighted Random Forest',
      'Precision': 1.0,
      'Recall': 0.005229734777736272}

```

```

[ ]: # get importance
importance = rnd_clf.feature_importances_
# summarize feature importance
for i,v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))
# plot feature importance
plt.bar([x for x in range(len(importance))], importance)
plt.show()

```

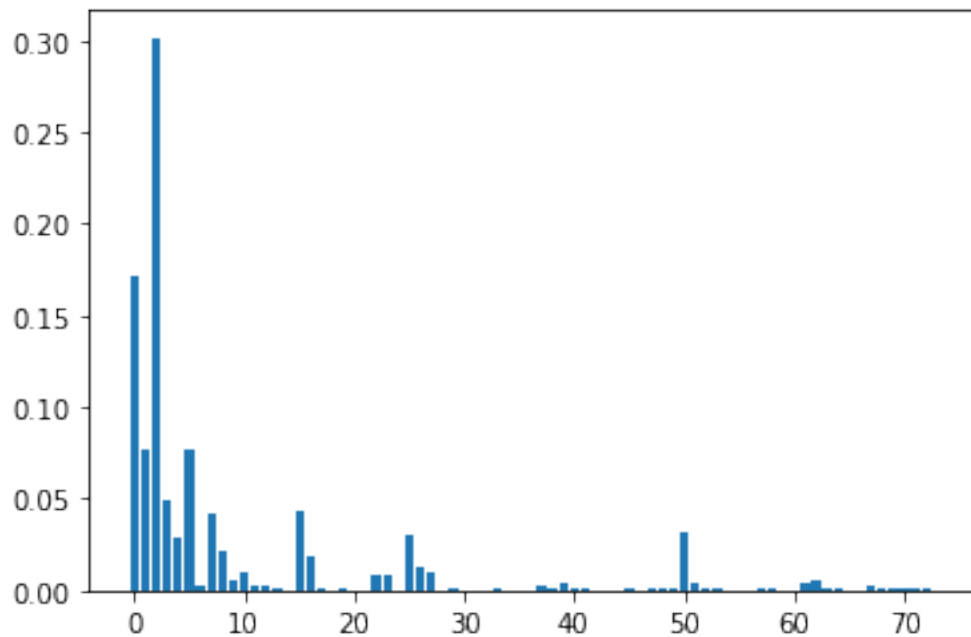
```

Feature: 0, Score: 0.17117
Feature: 1, Score: 0.07676
Feature: 2, Score: 0.30138
Feature: 3, Score: 0.04864
Feature: 4, Score: 0.02851
Feature: 5, Score: 0.07690
Feature: 6, Score: 0.00197
Feature: 7, Score: 0.04160
Feature: 8, Score: 0.02170
Feature: 9, Score: 0.00553
Feature: 10, Score: 0.01000
Feature: 11, Score: 0.00256
Feature: 12, Score: 0.00219
Feature: 13, Score: 0.00050
Feature: 14, Score: 0.00001
Feature: 15, Score: 0.04345
Feature: 16, Score: 0.01836
Feature: 17, Score: 0.00073
Feature: 18, Score: 0.00042
Feature: 19, Score: 0.00110
Feature: 20, Score: 0.00007
Feature: 21, Score: 0.00039
Feature: 22, Score: 0.00807

```

Feature: 23, Score: 0.00812  
Feature: 24, Score: 0.00024  
Feature: 25, Score: 0.03018  
Feature: 26, Score: 0.01237  
Feature: 27, Score: 0.00975  
Feature: 28, Score: 0.00007  
Feature: 29, Score: 0.00080  
Feature: 30, Score: 0.00014  
Feature: 31, Score: 0.00033  
Feature: 32, Score: 0.00005  
Feature: 33, Score: 0.00125  
Feature: 34, Score: 0.00005  
Feature: 35, Score: 0.00038  
Feature: 36, Score: 0.00006  
Feature: 37, Score: 0.00310  
Feature: 38, Score: 0.00072  
Feature: 39, Score: 0.00448  
Feature: 40, Score: 0.00052  
Feature: 41, Score: 0.00086  
Feature: 42, Score: 0.00025  
Feature: 43, Score: 0.00017  
Feature: 44, Score: 0.00021  
Feature: 45, Score: 0.00057  
Feature: 46, Score: 0.00018  
Feature: 47, Score: 0.00047  
Feature: 48, Score: 0.00075  
Feature: 49, Score: 0.00119  
Feature: 50, Score: 0.03232  
Feature: 51, Score: 0.00368  
Feature: 52, Score: 0.00110  
Feature: 53, Score: 0.00070  
Feature: 54, Score: 0.00042  
Feature: 55, Score: 0.00015  
Feature: 56, Score: 0.00020  
Feature: 57, Score: 0.00099  
Feature: 58, Score: 0.00061  
Feature: 59, Score: 0.00023  
Feature: 60, Score: 0.00044  
Feature: 61, Score: 0.00447  
Feature: 62, Score: 0.00549  
Feature: 63, Score: 0.00099  
Feature: 64, Score: 0.00063  
Feature: 65, Score: 0.00013  
Feature: 66, Score: 0.00036  
Feature: 67, Score: 0.00307  
Feature: 68, Score: 0.00085  
Feature: 69, Score: 0.00090  
Feature: 70, Score: 0.00173

Feature: 71, Score: 0.00062  
Feature: 72, Score: 0.00047  
Feature: 73, Score: 0.00021

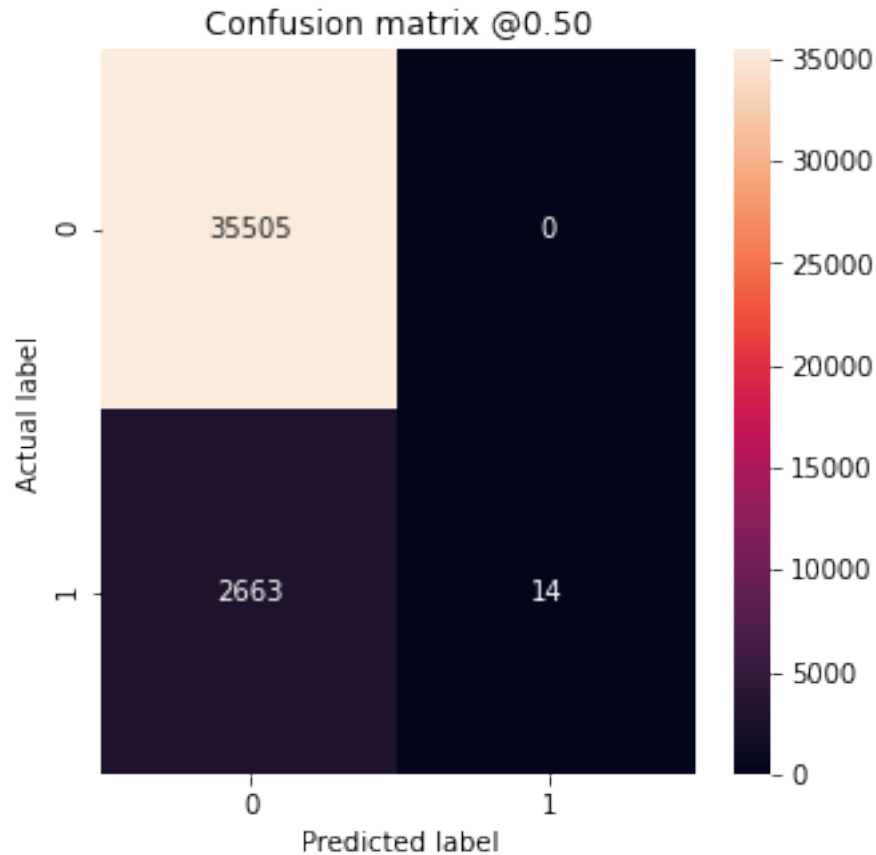


```
[ ]: train_predictions_baseline = rnd_clf.predict(train_features)
      test_predictions_baseline = rnd_clf.predict(test_features)

      plot_cm(test_labels, test_predictions_baseline)
```

f1 Score 0.010405053883314752

Irrelevant Documents Detected (True Negatives): 35505  
Irrelevant Documents Incorrectly Detected (False Positives): 0  
Relevant Documents Missed (False Negatives): 2663  
Relevant Documents Detected (True Positives): 14  
Total Relevant Documents: 2677



```
[ ]: test_predictions_baseline = rnd_clf.predict(test)
test_predictions_baseline = test_predictions_baseline.round(0)
test_predictions_baseline = test_predictions_baseline.astype(int)
test_predictions_baseline
```

```
[ ]: array([0, 0, 0, ..., 0, 0, 0])
```

```
[ ]: predictions_baseline= pd.DataFrame(test_predictions_baseline)
predictions_baseline['Id'] = predictions_baseline.index
predictions_baseline.rename(columns={ predictions_baseline.columns[0]: "psrel"↵
↵}, inplace = True)
predictions_baseline = predictions_baseline[['Id','psrel']]
predictions_baseline
```

```
[ ]:      Id  psrel
0      0      0
1      1      0
2      2      0
3      3      0
```



```

4          4          0
...      ...      ...
4995  4995      0
4996  4996      0
4997  4997      0
4998  4998      0
4999  4999      0

```

[5000 rows x 2 columns]

```
[ ]: predictions_baseline['psrel'].value_counts()
```

```
[ ]: 0      5000
      Name: psrel, dtype: int64
```

```
[ ]: predictions_baseline.to_csv('Random Forest colab predictions.csv', index=False)
```

Data is still currently too skewed for standard machine learning model to accurately predict true positives. Will use large batch sizes for neural network training to try and ensure some positive values are included per batch, as well as other methods detailed below

### 6.2.1 Three Layer Model Baseline

```
[ ]: EPOCHS = 100
      BATCH_SIZE = 2048

      early_stopping = tf.keras.callbacks.EarlyStopping(
          monitor='val_precision',
          verbose=1,
          patience=10,
          mode='max',
          restore_best_weights=True)

      # I've replaced val_auc for val_tp for monitor to see if getting more true
      # positives will improve performance
```

### Three Layer Model function

```
[ ]: # Three Layer Learning Model function
def build_three_layer_model(n_hidden=3, n_neurons=30, learning_rate=3e-4,
    metrics=METRICS, output_bias=None, input_shape=train_features.shape[1:]):
    if output_bias is not None:
        output_bias = tf.keras.initializers.Constant(output_bias)
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(RegularizedDense(n_neurons))
        model.add(keras.layers.BatchNormalization())
```

```

        model.add(keras.layers.Dropout(rate=0.3)),
        model.add(keras.layers.Dense(1, activation="sigmoid",
→bias_initializer=output_bias))

        optimizer = keras.optimizers.Nadam(lr=learning_rate)
        model.compile(loss="binary_crossentropy", optimizer=optimizer,
→metrics=metrics)

    return model

```

```
[ ]: # Create a default three layer model using the build_three_layer_model function
```

```

three_layer_keras_reg = keras.wrappers.scikit_learn.
→KerasRegressor(build_three_layer_model)

three_layer_model = build_three_layer_model()
three_layer_model.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 30)	2250
batch_normalization (Batch Normalization)	(None, 30)	120
dropout (Dropout)	(None, 30)	0
dense_1 (Dense)	(None, 30)	930
batch_normalization_1 (Batch Normalization)	(None, 30)	120
dropout_1 (Dropout)	(None, 30)	0
dense_2 (Dense)	(None, 30)	930
batch_normalization_2 (Batch Normalization)	(None, 30)	120
dropout_2 (Dropout)	(None, 30)	0
dense_3 (Dense)	(None, 1)	31

Total params: 4,501  
 Trainable params: 4,321  
 Non-trainable params: 180

```
[ ]: # Create onecycle for models
onecycle = OneCycleScheduler(math.ceil(len(train_labels) / BATCH_SIZE) * EPOCHS,
    ↳max_rate=0.05)
```

Test run to compare predictions on train set vs actual values (naive predictions)

```
[ ]: three_layer_model.predict(train_features[:10])
```

```
[ ]: array([[0.8346735 ],
          [0.27901396],
          [0.43054563],
          [0.3553014 ],
          [0.7721624 ],
          [0.781099  ],
          [0.5588717 ],
          [0.8045237 ],
          [0.96360946],
          [0.604001  ]], dtype=float32)
```

```
[ ]: train_labels[:10]
```

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

Every prediction was wrong so we'll try some methods to improve this

**Adjusting bias** Set correct initial bias to stop the algorithm from overfitting on the 0's

```
[ ]: # Current Loss

results = three_layer_model.evaluate(train_features, train_labels,
    ↳batch_size=BATCH_SIZE, verbose=0)
print("Loss: {:.4f}".format(results[0]))
```

Loss: 9.7782

```
[ ]: # What Loss should be, derived from equations
initial_bias = np.log([pos/neg])
initial_bias
```

```
[ ]: array([-2.7612479])
```

```
[ ]: pos/total
```

```
[ ]: 0.059454545454545454
```

```
[ ]: three_layer_model = build_three_layer_model(output_bias=initial_bias)
three_layer_model.predict(train_features[:10])
```

```
[ ]: array([[0.52869797],
           [0.03223044],
           [0.02136847],
           [0.08165911],
           [0.06045404],
           [0.13303357],
           [0.0526365 ],
           [0.04584333],
           [0.06245753],
           [0.0207577 ]], dtype=float32)
```

```
[ ]: # As a result the initial loss is much less reducing the time needed for the
      →neural network to initialise

results = three_layer_model.evaluate(train_features, train_labels,
      →batch_size=BATCH_SIZE, verbose=0)
print("Loss: {:.4f}".format(results[0]))
```

Loss: 8.9992

To make the various training runs more comparable, keep the initial model's weights in a checkpoint file, and load them into each model before training.

```
[ ]: initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
      three_layer_model.save_weights(initial_weights)
```

Before moving on, confirm quick that the careful bias initialisation actually helped.

Train the model for 20 epochs, with and without this careful initialisation, and compare the losses:

```
[ ]: # Without bias initialisation

three_layer_model = build_three_layer_model()
three_layer_model.load_weights(initial_weights)
three_layer_model.layers[-1].bias.assign([0.0])
zero_bias_history = three_layer_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=20,
    validation_data=(val_features, val_labels),
    verbose=0)
```

```
[ ]: # With bias initialisation

three_layer_model = build_three_layer_model()
three_layer_model.load_weights(initial_weights)
careful_bias_history = three_layer_model.fit(
    train_features,
```

```

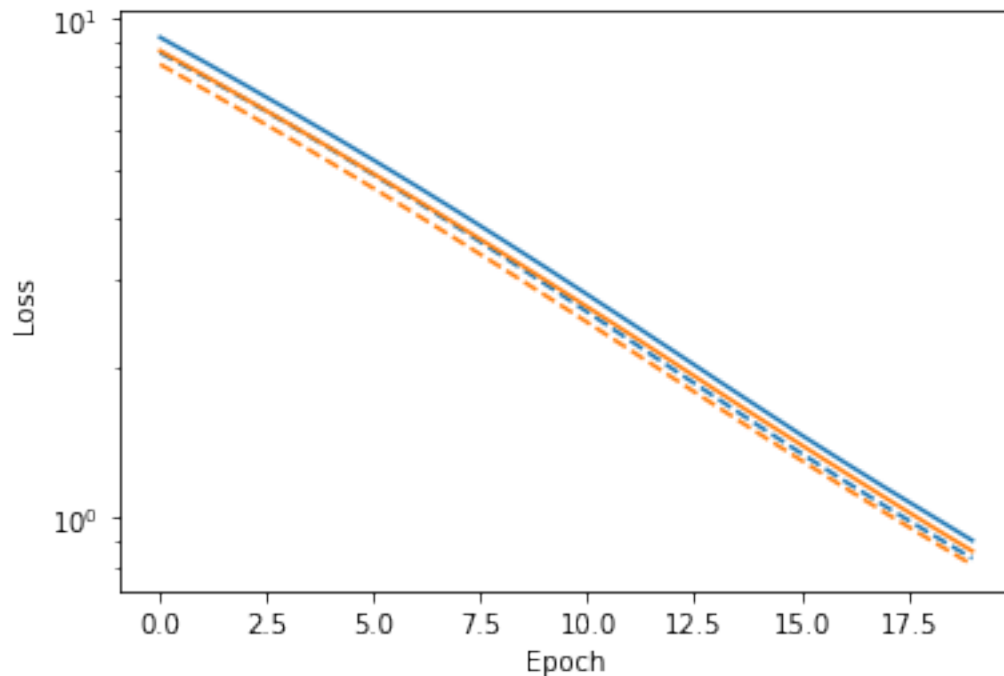
train_labels,
batch_size=BATCH_SIZE,
epochs=20,
validation_data=(val_features, val_labels),
verbose=0)

```

```

[ ]: plot_loss(zero_bias_history, "Zero Bias", 0) #blue
plot_loss(careful_bias_history, "Careful Bias", 1) #orange

```



### Train a model with adjusted biases

```

[ ]: three_layer_model = build_three_layer_model()
three_layer_model.load_weights(initial_weights)
baseline_history = three_layer_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels))

```

Epoch 1/100

60/60 [=====] - 6s 37ms/step - loss: 8.8345 - tp: 104.9344 - fp: 2083.7705 - tn: 85106.7213 - fn: 6681.0164 - accuracy: 0.9085 - precision: 0.0459 - recall: 0.0139 - auc: 0.5368 - val\_loss: 8.0972 - val\_tp:

0.0000e+00 - val\_fp: 25.0000 - val\_tn: 28336.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9277 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.5302

Epoch 2/100

60/60 [=====] - 1s 21ms/step - loss: 7.9342 - tp:  
77.3934 - fp: 1336.8852 - tn: 57450.1803 - fn: 4566.9836 - accuracy: 0.9055 -  
precision: 0.0558 - recall: 0.0178 - auc: 0.5139 - val\_loss: 7.2461 - val\_tp:  
0.0000e+00 - val\_fp: 2.0000 - val\_tn: 28359.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9284 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.5476

Epoch 3/100

60/60 [=====] - 1s 21ms/step - loss: 7.0935 - tp:  
45.2623 - fp: 863.1311 - tn: 57959.3279 - fn: 4563.7213 - accuracy: 0.9147 -  
precision: 0.0486 - recall: 0.0100 - auc: 0.5158 - val\_loss: 6.4770 - val\_tp:  
0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.5632

Epoch 4/100

60/60 [=====] - 1s 25ms/step - loss: 6.3396 - tp:  
26.7869 - fp: 526.2787 - tn: 58287.2459 - fn: 4591.1311 - accuracy: 0.9184 -  
precision: 0.0454 - recall: 0.0056 - auc: 0.5282 - val\_loss: 5.7780 - val\_tp:  
0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.5762

Epoch 5/100

60/60 [=====] - 1s 21ms/step - loss: 5.6476 - tp:  
22.0656 - fp: 351.6066 - tn: 58485.8033 - fn: 4571.9672 - accuracy: 0.9225 -  
precision: 0.0579 - recall: 0.0049 - auc: 0.5411 - val\_loss: 5.1424 - val\_tp:  
0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.5924

Epoch 6/100

60/60 [=====] - 1s 21ms/step - loss: 5.0231 - tp:  
9.8197 - fp: 236.3115 - tn: 58587.8033 - fn: 4597.5082 - accuracy: 0.9240 -  
precision: 0.0396 - recall: 0.0022 - auc: 0.5542 - val\_loss: 4.5689 - val\_tp:  
0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.6043

Epoch 7/100

60/60 [=====] - 1s 22ms/step - loss: 4.4665 - tp:  
5.8197 - fp: 144.8689 - tn: 58611.3770 - fn: 4669.3770 - accuracy: 0.9236 -  
precision: 0.0376 - recall: 0.0013 - auc: 0.5623 - val\_loss: 4.0511 - val\_tp:  
0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000 -  
val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.6135

Epoch 8/100

60/60 [=====] - 1s 21ms/step - loss: 3.9552 - tp:  
2.7541 - fp: 103.0656 - tn: 58686.6066 - fn: 4639.0164 - accuracy: 0.9254 -

precision: 0.0235 - recall: 5.4982e-04 - auc: 0.5736 - val\_loss: 3.5852 -  
val\_tp: 0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000  
- val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.6127

Epoch 9/100

60/60 [=====] - 1s 20ms/step - loss: 3.4984 - tp:  
2.2951 - fp: 58.1639 - tn: 58744.4262 - fn: 4626.5574 - accuracy: 0.9261 -  
precision: 0.0315 - recall: 4.2355e-04 - auc: 0.5794 - val\_loss: 3.1677 -  
val\_tp: 0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000  
- val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.6282

Epoch 10/100

60/60 [=====] - 1s 21ms/step - loss: 3.0874 - tp:  
3.7705 - fp: 26.8361 - tn: 58805.1311 - fn: 4595.7049 - accuracy: 0.9277 -  
precision: 0.1080 - recall: 7.2174e-04 - auc: 0.5862 - val\_loss: 2.7956 -  
val\_tp: 0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000  
- val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.6241

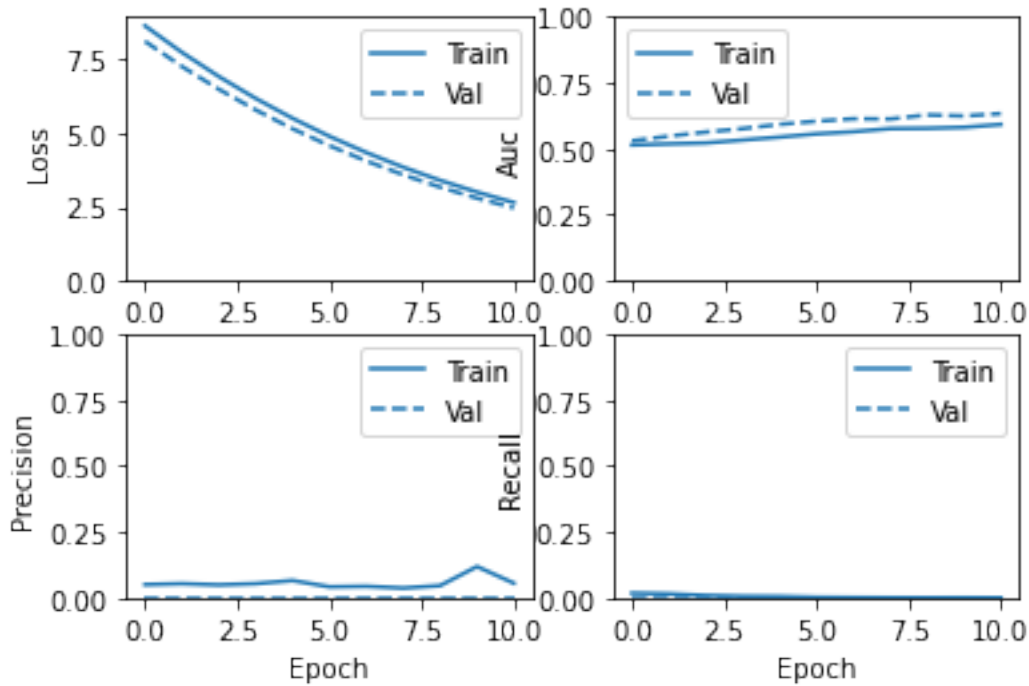
Epoch 11/100

60/60 [=====] - 1s 21ms/step - loss: 2.7247 - tp:  
2.0492 - fp: 27.0656 - tn: 58828.8197 - fn: 4573.5082 - accuracy: 0.9276 -  
precision: 0.0687 - recall: 4.0854e-04 - auc: 0.5892 - val\_loss: 2.4641 -  
val\_tp: 0.0000e+00 - val\_fp: 0.0000e+00 - val\_tn: 28361.0000 - val\_fn: 2184.0000  
- val\_accuracy: 0.9285 - val\_precision: 0.0000e+00 - val\_recall: 0.0000e+00 -  
val\_auc: 0.6331

Restoring model weights from the end of the best epoch.

Epoch 00011: early stopping

```
[ ]: # produce plots of model's accuracy and loss on the training and validation set  
     ↳ also includes some other metrics  
plot_metrics(baseline_history)
```



```
[ ]: train_predictions_baseline = three_layer_model.predict(train_features,
    ↳batch_size=BATCH_SIZE)
test_predictions_baseline = three_layer_model.predict(test_features,
    ↳batch_size=BATCH_SIZE)

# Print out scores for test values and confusion matrix

baseline_results = three_layer_model.evaluate(test_features, test_labels,
    batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(three_layer_model.metrics_names, baseline_results):
    print(name, ': ', value)
print()

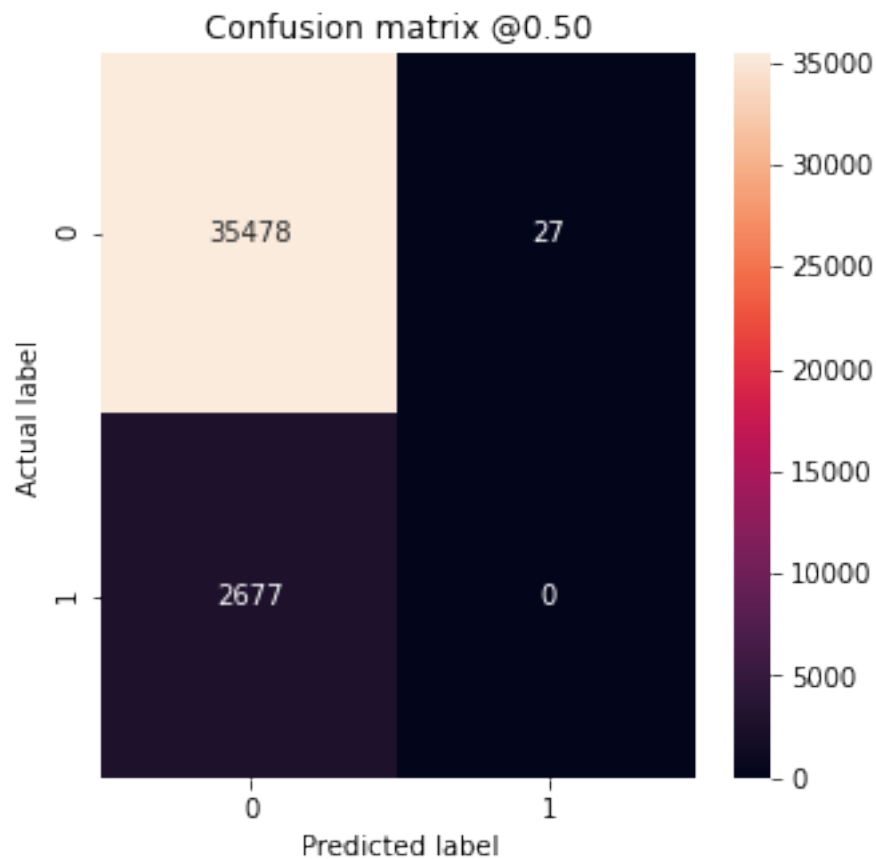
plot_cm(test_labels, test_predictions_baseline)
```

```
loss : 8.092643737792969
tp : 0.0
fp : 27.0
tn : 35478.0
fn : 2677.0
accuracy : 0.9291812777519226
precision : 0.0
recall : 0.0
auc : 0.5336455702781677
```



f1 Score 0.0

Irrelevant Documents Detected (True Negatives): 35478  
Irrelevant Documents Incorrectly Detected (False Positives): 27  
Relevant Documents Missed (False Negatives): 2677  
Relevant Documents Detected (True Positives): 0  
Total Relevant Documents: 2677



```
[ ]: (three_layer_model.metrics_names[0], baseline_results[0])
```

```
[ ]: ('loss', 8.092643737792969)
```

```
[ ]: pred_labels = three_layer_model.predict(test_features)
     pred_labels = pred_labels.round(0)
     pred_labels = pred_labels.astype(int)

     tl_bias = {'Model': 'Adjusted Bias Three Layer Model',
               'Loss': baseline_results[0],
               'Accuracy': accuracy_score(test_labels, pred_labels),
               'Precision': precision_score(test_labels, pred_labels),
```

```
'Recall': recall_score(test_labels, pred_labels),  
'F1 Score': f1_score(test_labels, pred_labels)}
```

```
tl_bias
```

```
[ ]: {'Accuracy': 0.9291812896129066,  
      'F1 Score': 0.0,  
      'Loss': 8.092643737792969,  
      'Model': 'Adjusted Bias Three Layer Model',  
      'Precision': 0.0,  
      'Recall': 0.0}
```

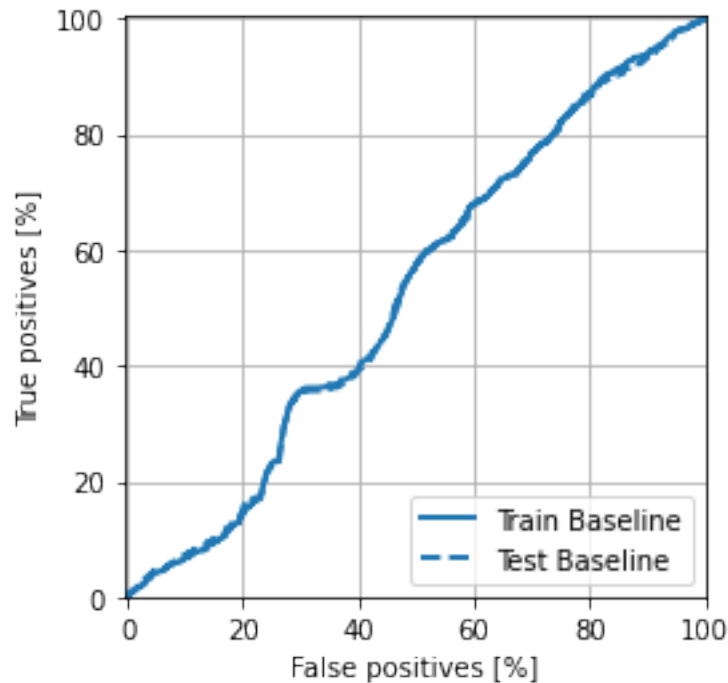
```
[ ]: !mkdir -p saved_model  
three_layer_model.save('saved_model/three_layer_adjusted_bias_model')
```

```
INFO:tensorflow:Assets written to:  
saved_model/three_layer_adjusted_bias_model/assets
```

If the model had predicted everything perfectly, this would be a diagonal matrix where values off the main diagonal, indicating incorrect predictions, would be zero. In this case the matrix shows that you have relatively few false positives, meaning that there were relatively few irrelevant documents that were incorrectly flagged. However, you would likely want to have even fewer false negatives despite the cost of increasing the number of false positives.

```
[ ]: # Now plot the ROC. This plot is useful because it shows, at a glance, the range  
      ↳ of performance the model can reach just by tuning the output threshold.  
  
plot_roc("Train Baseline", train_labels, train_predictions_baseline, ↳  
      ↳ color=colors[0])  
plot_roc("Test Baseline", test_labels, test_predictions_baseline, ↳  
      ↳ color=colors[0], linestyle='--')  
plt.legend(loc='lower right')
```

```
[ ]: <matplotlib.legend.Legend at 0x7f750efe7f10>
```



**Tweaking Class Weights** The goal is to identify relevant documents, but you don't have very many of those positive samples to work with, so you would want to have the classifier heavily weight the few examples that are available. You can do this by passing Keras weights for each class through a parameter. These will cause the model to “pay more attention” to examples from an under-represented class.

```
[ ]: # Scaling by total/2 helps keep the loss to a similar magnitude.
      # The sum of the weights of all examples stays the same.
      weight_for_0 = (1 / neg)*(total)/2.0
      weight_for_1 = (1 / pos)*(total)/2.0

      class_weight = {0: weight_for_0, 1: weight_for_1}

      print('Weight for class 0: {:.2f}'.format(weight_for_0))
      print('Weight for class 1: {:.2f}'.format(weight_for_1))
```

Weight for class 0: 0.53

Weight for class 1: 8.41

**Train a model with class weights** Now try re-training and evaluating the model with class weights to see how that affects the predictions.

Note: Using `class_weights` changes the range of the loss. This may affect the stability of the training depending on the optimizer. Optimizers whose step size is dependent on the magnitude of the gradient, like `optimizers.SGD`, may fail. The optimizer used here, `optimizers.Nadam`, is

unaffected by the scaling change. Also note that because of the weighting, the total losses are not comparable between the two models.

```
[ ]: weighted_three_layer_model = build_three_layer_model()
weighted_three_layer_model.load_weights(initial_weights)

weighted_history = weighted_three_layer_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels),
    class_weight=class_weight)
```

Epoch 1/100

60/60 [=====] - 6s 43ms/step - loss: 10.3057 - tp: 138.9672 - fp: 2236.7869 - tn: 92102.2459 - fn: 7135.4426 - accuracy: 0.9097 - precision: 0.0545 - recall: 0.0169 - auc: 0.5273 - val\_loss: 8.2965 - val\_tp: 22.0000 - val\_fp: 176.0000 - val\_tn: 28185.0000 - val\_fn: 2162.0000 - val\_accuracy: 0.9235 - val\_precision: 0.1111 - val\_recall: 0.0101 - val\_auc: 0.6183

Epoch 2/100

60/60 [=====] - 1s 20ms/step - loss: 9.3726 - tp: 321.5574 - fp: 3239.5082 - tn: 55544.8361 - fn: 4325.5410 - accuracy: 0.8838 - precision: 0.0876 - recall: 0.0626 - auc: 0.5873 - val\_loss: 7.6420 - val\_tp: 12.0000 - val\_fp: 576.0000 - val\_tn: 27785.0000 - val\_fn: 2172.0000 - val\_accuracy: 0.9100 - val\_precision: 0.0204 - val\_recall: 0.0055 - val\_auc: 0.6420

Epoch 3/100

60/60 [=====] - 1s 21ms/step - loss: 8.5441 - tp: 668.1148 - fp: 5703.0000 - tn: 53098.1967 - fn: 3962.1311 - accuracy: 0.8498 - precision: 0.1023 - recall: 0.1370 - auc: 0.6119 - val\_loss: 7.0059 - val\_tp: 27.0000 - val\_fp: 924.0000 - val\_tn: 27437.0000 - val\_fn: 2157.0000 - val\_accuracy: 0.8991 - val\_precision: 0.0284 - val\_recall: 0.0124 - val\_auc: 0.6574

Epoch 4/100

60/60 [=====] - 1s 21ms/step - loss: 7.7920 - tp: 957.4754 - fp: 7838.1967 - tn: 50982.0328 - fn: 3653.7377 - accuracy: 0.8212 - precision: 0.1094 - recall: 0.2035 - auc: 0.6276 - val\_loss: 6.4118 - val\_tp: 148.0000 - val\_fp: 1465.0000 - val\_tn: 26896.0000 - val\_fn: 2036.0000 - val\_accuracy: 0.8854 - val\_precision: 0.0918 - val\_recall: 0.0678 - val\_auc: 0.6702

Epoch 5/100

60/60 [=====] - 1s 20ms/step - loss: 7.1512 - tp: 1172.9344 - fp: 9485.6557 - tn: 49293.3115 - fn: 3479.5410 - accuracy: 0.7963 - precision: 0.1108 - recall: 0.2505 - auc: 0.6321 - val\_loss: 5.8646 - val\_tp: 440.0000 - val\_fp: 2641.0000 - val\_tn: 25720.0000 - val\_fn: 1744.0000 -

val\_accuracy: 0.8564 - val\_precision: 0.1428 - val\_recall: 0.2015 - val\_auc: 0.6727

Epoch 6/100

60/60 [=====] - 1s 21ms/step - loss: 6.5053 - tp: 1368.3443 - fp: 10874.6885 - tn: 47922.9836 - fn: 3265.4262 - accuracy: 0.7790 - precision: 0.1128 - recall: 0.2915 - auc: 0.6419 - val\_loss: 5.3683 - val\_tp: 518.0000 - val\_fp: 3545.0000 - val\_tn: 24816.0000 - val\_fn: 1666.0000 - val\_accuracy: 0.8294 - val\_precision: 0.1275 - val\_recall: 0.2372 - val\_auc: 0.6745

Epoch 7/100

60/60 [=====] - 1s 21ms/step - loss: 5.9404 - tp: 1567.7049 - fp: 12564.8033 - tn: 46269.6721 - fn: 3029.2623 - accuracy: 0.7556 - precision: 0.1128 - recall: 0.3401 - auc: 0.6421 - val\_loss: 4.9131 - val\_tp: 786.0000 - val\_fp: 4769.0000 - val\_tn: 23592.0000 - val\_fn: 1398.0000 - val\_accuracy: 0.7981 - val\_precision: 0.1415 - val\_recall: 0.3599 - val\_auc: 0.6808

Epoch 8/100

60/60 [=====] - 1s 20ms/step - loss: 5.4045 - tp: 1724.4754 - fp: 13908.8852 - tn: 44912.4262 - fn: 2885.6557 - accuracy: 0.7374 - precision: 0.1108 - recall: 0.3703 - auc: 0.6453 - val\_loss: 4.4966 - val\_tp: 892.0000 - val\_fp: 5537.0000 - val\_tn: 22824.0000 - val\_fn: 1292.0000 - val\_accuracy: 0.7764 - val\_precision: 0.1387 - val\_recall: 0.4084 - val\_auc: 0.6838

Epoch 9/100

60/60 [=====] - 1s 21ms/step - loss: 4.9015 - tp: 1862.5082 - fp: 15216.6393 - tn: 43654.0820 - fn: 2698.2131 - accuracy: 0.7194 - precision: 0.1094 - recall: 0.4115 - auc: 0.6501 - val\_loss: 4.1190 - val\_tp: 1470.0000 - val\_fp: 11500.0000 - val\_tn: 16861.0000 - val\_fn: 714.0000 - val\_accuracy: 0.6001 - val\_precision: 0.1133 - val\_recall: 0.6731 - val\_auc: 0.6881

Epoch 10/100

60/60 [=====] - 1s 22ms/step - loss: 4.4885 - tp: 1999.1148 - fp: 16633.3934 - tn: 42198.9672 - fn: 2599.9672 - accuracy: 0.6988 - precision: 0.1054 - recall: 0.4240 - auc: 0.6432 - val\_loss: 3.7676 - val\_tp: 1533.0000 - val\_fp: 12303.0000 - val\_tn: 16058.0000 - val\_fn: 651.0000 - val\_accuracy: 0.5759 - val\_precision: 0.1108 - val\_recall: 0.7019 - val\_auc: 0.6908

Epoch 11/100

60/60 [=====] - 1s 21ms/step - loss: 4.1046 - tp: 2103.6557 - fp: 17569.8033 - tn: 41252.3934 - fn: 2505.5902 - accuracy: 0.6848 - precision: 0.1070 - recall: 0.4534 - auc: 0.6456 - val\_loss: 3.4532 - val\_tp: 1585.0000 - val\_fp: 12741.0000 - val\_tn: 15620.0000 - val\_fn: 599.0000 - val\_accuracy: 0.5633 - val\_precision: 0.1106 - val\_recall: 0.7257 - val\_auc: 0.6917

Epoch 12/100

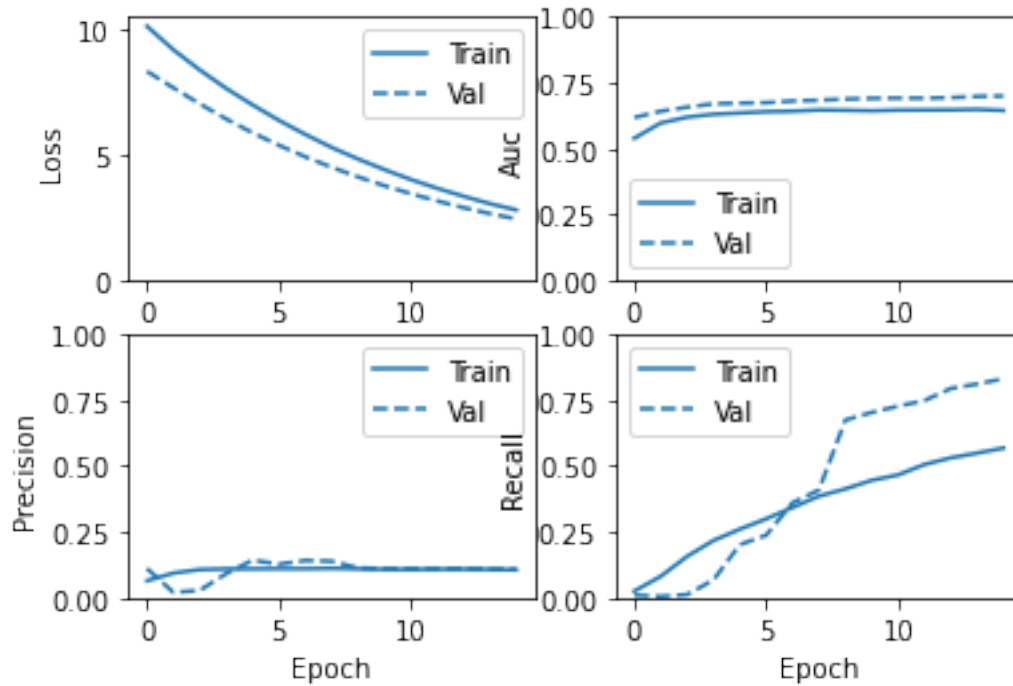
60/60 [=====] - 1s 21ms/step - loss: 3.7452 - tp: 2298.8689 - fp: 18923.8852 - tn: 39899.4590 - fn: 2309.2295 - accuracy: 0.6659 - precision: 0.1072 - recall: 0.4931 - auc: 0.6452 - val\_loss: 3.1652 - val\_tp:

```

1631.0000 - val_fp: 13142.0000 - val_tn: 15219.0000 - val_fn: 553.0000 -
val_accuracy: 0.5516 - val_precision: 0.1104 - val_recall: 0.7468 - val_auc:
0.6913
Epoch 13/100
60/60 [=====] - 1s 21ms/step - loss: 3.4217 - tp:
2427.4754 - fp: 19871.7377 - tn: 38961.5410 - fn: 2170.6885 - accuracy: 0.6527 -
precision: 0.1089 - recall: 0.5272 - auc: 0.6472 - val_loss: 2.8987 - val_tp:
1731.0000 - val_fp: 13816.0000 - val_tn: 14545.0000 - val_fn: 453.0000 -
val_accuracy: 0.5329 - val_precision: 0.1113 - val_recall: 0.7926 - val_auc:
0.6937
Epoch 14/100
60/60 [=====] - 1s 21ms/step - loss: 3.1272 - tp:
2495.2295 - fp: 20675.4754 - tn: 38134.8525 - fn: 2125.8852 - accuracy: 0.6405 -
precision: 0.1075 - recall: 0.5385 - auc: 0.6458 - val_loss: 2.6647 - val_tp:
1768.0000 - val_fp: 14517.0000 - val_tn: 13844.0000 - val_fn: 416.0000 -
val_accuracy: 0.5111 - val_precision: 0.1086 - val_recall: 0.8095 - val_auc:
0.6980
Epoch 15/100
60/60 [=====] - 1s 21ms/step - loss: 2.8622 - tp:
2573.6393 - fp: 21872.7377 - tn: 36985.1803 - fn: 1999.8852 - accuracy: 0.6243 -
precision: 0.1042 - recall: 0.5576 - auc: 0.6425 - val_loss: 2.4529 - val_tp:
1812.0000 - val_fp: 14911.0000 - val_tn: 13450.0000 - val_fn: 372.0000 -
val_accuracy: 0.4997 - val_precision: 0.1084 - val_recall: 0.8297 - val_auc:
0.6995
Restoring model weights from the end of the best epoch.
Epoch 00015: early stopping

```

```
[ ]: plot_metrics(weighted_history)
```



```
[ ]: train_predictions_weighted = weighted_three_layer_model.predict(train_features,
    ↳batch_size=BATCH_SIZE)
test_predictions_weighted = weighted_three_layer_model.predict(test_features,
    ↳batch_size=BATCH_SIZE)
```

```
[ ]: weighted_results = weighted_three_layer_model.evaluate(test_features,
    ↳test_labels,
                                batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(weighted_three_layer_model.metrics_names,
    ↳weighted_results):
    print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_weighted)
```

```
loss : 5.860197067260742
tp : 537.0
fp : 3204.0
tn : 32301.0
fn : 2140.0
accuracy : 0.8600387573242188
precision : 0.1435445100069046
recall : 0.20059768855571747
auc : 0.6826937198638916
```

f1 Score 0.16734185104393892

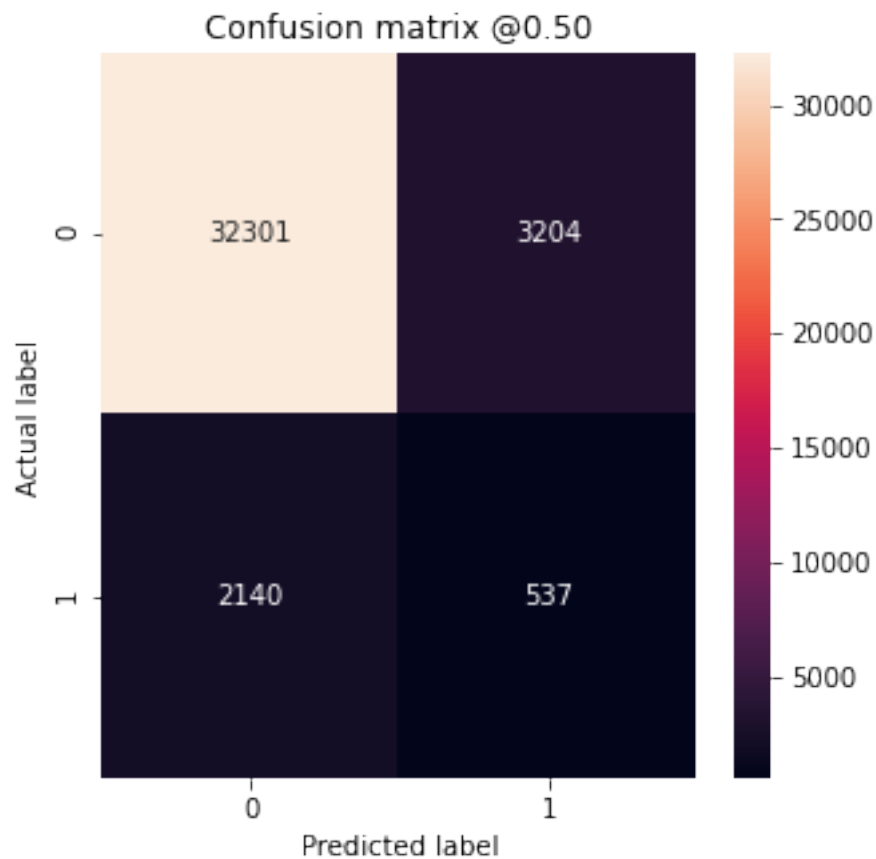
Irrelevant Documents Detected (True Negatives): 32301

Irrelevant Documents Incorrectly Detected (False Positives): 3204

Relevant Documents Missed (False Negatives): 2140

Relevant Documents Detected (True Positives): 537

Total Relevant Documents: 2677



```
[ ]: pred_labels = weighted_three_layer_model.predict(test_features)
pred_labels = pred_labels.round(0)
pred_labels = pred_labels.astype(int)

tl_wt = {'Model': 'Adjusted Weights Three Layer Model',
        'Loss': weighted_results[0],
        'Accuracy': accuracy_score(test_labels, pred_labels),
        'Precision': precision_score(test_labels, pred_labels),
        'Recall': recall_score(test_labels, pred_labels),
        'F1 Score': f1_score(test_labels, pred_labels)}
```



```
tl_wt
```

```
[ ]: {'Accuracy': 0.8600387617201822,  
      'F1 Score': 0.16734185104393892,  
      'Loss': 5.860197067260742,  
      'Model': 'Adjusted Weights Three Layer Model',  
      'Precision': 0.14354450681635927,  
      'Recall': 0.20059768397459843}
```

```
[ ]: test_predictions = weighted_three_layer_model.predict(test)
```

```
[ ]: test_predictions = test_predictions.round(0)  
test_predictions = test_predictions.astype(int)  
test_predictions
```

```
[ ]: array([[0],  
           [0],  
           [0],  
           ...,  
           [0],  
           [0],  
           [0]])
```

```
[ ]: predictions= pd.DataFrame(test_predictions)  
predictions['Id'] = predictions.index  
predictions.rename(columns={ predictions.columns[0]: "psrel" }, inplace = True)  
predictions = predictions[['Id', 'psrel']]  
predictions
```

```
[ ]:      Id  psrel  
0      0      0  
1      1      0  
2      2      0  
3      3      0  
4      4      0  
...    ...    ...  
4995  4995      0  
4996  4996      0  
4997  4997      0  
4998  4998      0  
4999  4999      0
```

```
[5000 rows x 2 columns]
```

```
[ ]: predictions['psrel'].value_counts()
```

```
[ ]: 0    4711
      1     289
      Name: psrel, dtype: int64
```

```
[ ]: predictions.to_csv('adjusted class weights colab predictions class weights.csv',
      →index=False)
```

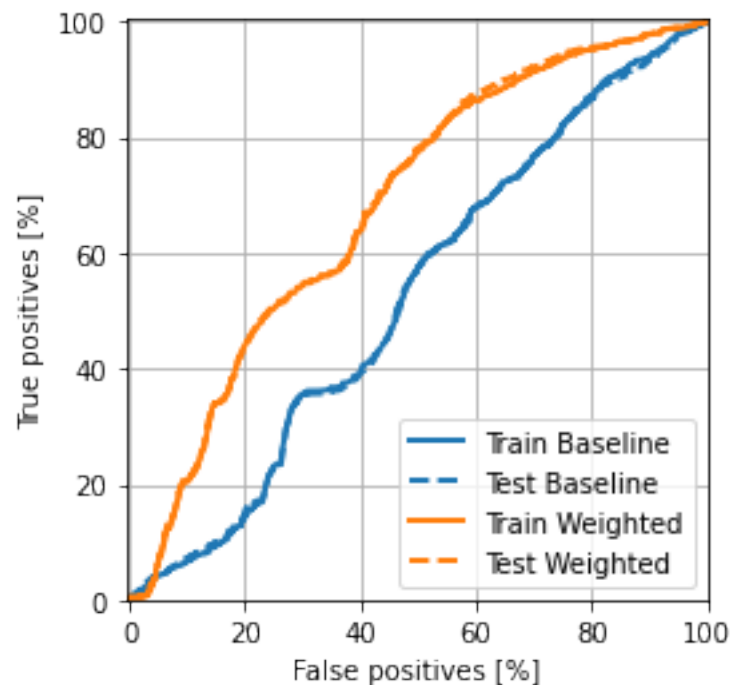
Here you can see that with class weights the accuracy and precision are lower because there are more false positives, but conversely the recall and AUC are higher because the model also found more true positives. Despite having lower accuracy, this model has higher recall.

```
[ ]: plot_roc("Train Baseline", train_labels, train_predictions_baseline,
      →color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline,
      →color=colors[0], linestyle='--')

plot_roc("Train Weighted", train_labels, train_predictions_weighted,
      →color=colors[1])
plot_roc("Test Weighted", test_labels, test_predictions_weighted,
      →color=colors[1], linestyle='--')

plt.legend(loc='lower right')
```

```
[ ]: <matplotlib.legend.Legend at 0x7f7510aed510>
```



```
[ ]: !mkdir -p saved_model
weighted_three_layer_model.save('saved_model/weighted_three_layer_model')
```

INFO:tensorflow:Assets written to: saved\_model/weighted\_three\_layer\_model/assets

**Oversample the minority class** A related approach would be to resample the dataset by oversampling the minority class. This helps the model to recognise what a relevant document looks like

```
[ ]: pos_features = train_features[bool_train_labels]
neg_features = train_features[~bool_train_labels]

pos_labels = train_labels[bool_train_labels]
neg_labels = train_labels[~bool_train_labels]
```

```
[ ]: # Use Numpy to generate new 'relevant' samples
ids = np.arange(len(pos_features))
choices = np.random.choice(ids, len(neg_features))

res_pos_features = pos_features[choices]
res_pos_labels = pos_labels[choices]

res_pos_features.shape
```

```
[ ]: (113299, 74)
```

```
[ ]: resampled_features = np.concatenate([res_pos_features, neg_features], axis=0)
resampled_labels = np.concatenate([res_pos_labels, neg_labels], axis=0)

order = np.arange(len(resampled_labels))
np.random.shuffle(order)
resampled_features = resampled_features[order]
resampled_labels = resampled_labels[order]

resampled_features.shape
```

```
[ ]: (226598, 74)
```

```
[ ]: # Use tf.data to merge positive and negative dataframes
BUFFER_SIZE = 100000

def make_ds(features, labels):
    ds = tf.data.Dataset.from_tensor_slices((features, labels)).cache()
    ds = ds.shuffle(BUFFER_SIZE).repeat()
    return ds

pos_ds = make_ds(pos_features, pos_labels)
neg_ds = make_ds(neg_features, neg_labels)
```

```
[ ]: # Each dataset provides (feature, label) pairs:
for features, label in pos_ds.take(1):
    print("Features:\n", features.numpy())
    print()
    print("Label: ", label.numpy())
```

Features:

```
[-0.31494735 -0.22480333 -0.35316283  0.5158269  -0.47299417  0.82892249
-0.10434731 -0.45014672 -0.54346398 -0.31365016 -0.43049886 -0.1641563
-0.069837   -0.06791763 -0.02774807  1.01496725 -0.49997698 -0.07420054
-0.06537769 -0.1215902  -0.02774807 -0.04427166 -0.24920441  0.25407715
-0.01458931 -0.73106047 -0.26347516 -0.30608251 -0.02932804 -0.05977517
-0.02478367 -0.04635734 -0.01311139 -0.05929057 -0.01786913 -0.04944702
-0.02759997 -0.07536263 -0.08439168 -0.10410658 -0.07216581 -0.10566194
-0.06882387 -0.03575538 -0.05541193 -0.06989614 -0.01213864 -0.07864309
-0.03470738 -0.10366385  0.90503443 -0.15992708 -0.11977513 -0.06742941
-0.04818352 -0.05132459 -0.05313653 -0.05915139 -0.10924712 -0.07001429
-0.06512468 -0.269587   -0.26484034 -0.13666075 -0.06512468 -0.0164368
-0.06455184 -0.24527183 -0.12591953 -0.12033636 -0.14459002 -0.09408335
-0.09670439 -0.0551139 ]
```

Label: 1

```
[ ]: resampled_ds = tf.data.experimental.sample_from_datasets([pos_ds, neg_ds],
    ↳weights=[0.5, 0.5])
resampled_ds = resampled_ds.batch(BATCH_SIZE).prefetch(2)
```

```
[ ]: # To use this dataset, you'll need the number of steps per epoch. Define as
    ↳number of batches to see each value of 1 once
resampled_steps_per_epoch = np.ceil(2.0*neg/BATCH_SIZE)
resampled_steps_per_epoch
```

```
[ ]: 31.0
```

**Train a Model with Oversampled Data** Training the model with the resampled data set instead of using class weights to see how these methods compare.

Note: Because the data was balanced by replicating the positive examples, the total dataset size is larger, and each epoch runs for more training steps.

```
[ ]: resampled_three_layer_model = build_three_layer_model()
resampled_three_layer_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_three_layer_model.layers[-1]
output_layer.bias.assign([0])

val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).cache()
```

```

val_ds = val_ds.batch(BATCH_SIZE).prefetch(2)

resampled_history = resampled_three_layer_model.fit(
    resampled_ds,
    epochs=EPOCHS,
    steps_per_epoch=resampled_steps_per_epoch,
    callbacks=[early_stopping],
    validation_data=val_ds)

```

Epoch 1/100

31/31 [=====] - 5s 68ms/step - loss: 9.5194 - tp: 9385.6562 - fp: 11888.5000 - tn: 40592.1250 - fn: 10043.7188 - accuracy: 0.7096 - precision: 0.4165 - recall: 0.4618 - auc: 0.6823 - val\_loss: 9.0466 - val\_tp: 1393.0000 - val\_fp: 16330.0000 - val\_tn: 12031.0000 - val\_fn: 791.0000 - val\_accuracy: 0.4395 - val\_precision: 0.0786 - val\_recall: 0.6378 - val\_auc: 0.5335

Epoch 2/100

31/31 [=====] - 1s 36ms/step - loss: 9.0495 - tp: 9315.6250 - fp: 8538.5312 - tn: 8269.3438 - fn: 7604.5000 - accuracy: 0.5203 - precision: 0.5211 - recall: 0.5487 - auc: 0.5220 - val\_loss: 8.6236 - val\_tp: 1523.0000 - val\_fp: 17169.0000 - val\_tn: 11192.0000 - val\_fn: 661.0000 - val\_accuracy: 0.4163 - val\_precision: 0.0815 - val\_recall: 0.6973 - val\_auc: 0.5629

Epoch 3/100

31/31 [=====] - 1s 37ms/step - loss: 8.5716 - tp: 9468.8750 - fp: 8530.3750 - tn: 8470.3750 - fn: 7258.3750 - accuracy: 0.5313 - precision: 0.5252 - recall: 0.5648 - auc: 0.5374 - val\_loss: 8.1928 - val\_tp: 1589.0000 - val\_fp: 17250.0000 - val\_tn: 11111.0000 - val\_fn: 595.0000 - val\_accuracy: 0.4158 - val\_precision: 0.0843 - val\_recall: 0.7276 - val\_auc: 0.5799

Epoch 4/100

31/31 [=====] - 1s 37ms/step - loss: 8.1289 - tp: 9582.0312 - fp: 8414.1562 - tn: 8571.9375 - fn: 7159.8750 - accuracy: 0.5391 - precision: 0.5327 - recall: 0.5734 - auc: 0.5444 - val\_loss: 7.7765 - val\_tp: 1639.0000 - val\_fp: 17319.0000 - val\_tn: 11042.0000 - val\_fn: 545.0000 - val\_accuracy: 0.4152 - val\_precision: 0.0865 - val\_recall: 0.7505 - val\_auc: 0.5918

Epoch 5/100

31/31 [=====] - 1s 37ms/step - loss: 7.7070 - tp: 9536.2812 - fp: 8427.2188 - tn: 8619.8125 - fn: 7144.6875 - accuracy: 0.5350 - precision: 0.5265 - recall: 0.5685 - auc: 0.5444 - val\_loss: 7.3784 - val\_tp: 1688.0000 - val\_fp: 17189.0000 - val\_tn: 11172.0000 - val\_fn: 496.0000 - val\_accuracy: 0.4210 - val\_precision: 0.0894 - val\_recall: 0.7729 - val\_auc: 0.5990

Epoch 6/100

31/31 [=====] - 1s 38ms/step - loss: 7.2957 - tp: 9867.9062 - fp: 8335.7812 - tn: 8493.0000 - fn: 7031.3125 - accuracy: 0.5445 - precision: 0.5425 - recall: 0.5835 - auc: 0.5534 - val\_loss: 7.0061 - val\_tp:

1715.0000 - val\_fp: 17391.0000 - val\_tn: 10970.0000 - val\_fn: 469.0000 -  
val\_accuracy: 0.4153 - val\_precision: 0.0898 - val\_recall: 0.7853 - val\_auc:  
0.6057

Epoch 7/100

31/31 [=====] - 1s 38ms/step - loss: 6.9091 - tp:  
10166.5625 - fp: 8351.0000 - tn: 8438.7188 - fn: 6771.7188 - accuracy: 0.5538 -  
precision: 0.5524 - recall: 0.6014 - auc: 0.5630 - val\_loss: 6.6473 - val\_tp:  
1725.0000 - val\_fp: 17333.0000 - val\_tn: 11028.0000 - val\_fn: 459.0000 -  
val\_accuracy: 0.4175 - val\_precision: 0.0905 - val\_recall: 0.7898 - val\_auc:  
0.6127

Epoch 8/100

31/31 [=====] - 1s 37ms/step - loss: 6.5414 - tp:  
10254.6562 - fp: 8428.7812 - tn: 8435.5625 - fn: 6609.0000 - accuracy: 0.5530 -  
precision: 0.5480 - recall: 0.6065 - auc: 0.5632 - val\_loss: 6.2940 - val\_tp:  
1722.0000 - val\_fp: 17203.0000 - val\_tn: 11158.0000 - val\_fn: 462.0000 -  
val\_accuracy: 0.4217 - val\_precision: 0.0910 - val\_recall: 0.7885 - val\_auc:  
0.6174

Epoch 9/100

31/31 [=====] - 1s 37ms/step - loss: 6.1925 - tp:  
10399.1562 - fp: 8385.6250 - tn: 8439.4375 - fn: 6503.7812 - accuracy: 0.5580 -  
precision: 0.5523 - recall: 0.6144 - auc: 0.5690 - val\_loss: 5.9618 - val\_tp:  
1728.0000 - val\_fp: 17142.0000 - val\_tn: 11219.0000 - val\_fn: 456.0000 -  
val\_accuracy: 0.4239 - val\_precision: 0.0916 - val\_recall: 0.7912 - val\_auc:  
0.6230

Epoch 10/100

31/31 [=====] - 1s 36ms/step - loss: 5.8587 - tp:  
10563.5312 - fp: 8395.1562 - tn: 8451.0000 - fn: 6318.3125 - accuracy: 0.5644 -  
precision: 0.5569 - recall: 0.6264 - auc: 0.5739 - val\_loss: 5.6436 - val\_tp:  
1757.0000 - val\_fp: 17073.0000 - val\_tn: 11288.0000 - val\_fn: 427.0000 -  
val\_accuracy: 0.4271 - val\_precision: 0.0933 - val\_recall: 0.8045 - val\_auc:  
0.6279

Epoch 11/100

31/31 [=====] - 1s 37ms/step - loss: 5.5368 - tp:  
10636.7500 - fp: 8403.7812 - tn: 8536.0938 - fn: 6151.3750 - accuracy: 0.5693 -  
precision: 0.5588 - recall: 0.6346 - auc: 0.5814 - val\_loss: 5.3398 - val\_tp:  
1751.0000 - val\_fp: 16861.0000 - val\_tn: 11500.0000 - val\_fn: 433.0000 -  
val\_accuracy: 0.4338 - val\_precision: 0.0941 - val\_recall: 0.8017 - val\_auc:  
0.6331

Epoch 12/100

31/31 [=====] - 1s 38ms/step - loss: 5.2361 - tp:  
10780.5625 - fp: 8272.3438 - tn: 8621.6875 - fn: 6053.4062 - accuracy: 0.5754 -  
precision: 0.5653 - recall: 0.6392 - auc: 0.5898 - val\_loss: 5.0553 - val\_tp:  
1746.0000 - val\_fp: 16767.0000 - val\_tn: 11594.0000 - val\_fn: 438.0000 -  
val\_accuracy: 0.4367 - val\_precision: 0.0943 - val\_recall: 0.7995 - val\_auc:  
0.6377

Epoch 13/100

31/31 [=====] - 1s 38ms/step - loss: 4.9485 - tp:  
10986.7812 - fp: 8357.8125 - tn: 8491.5938 - fn: 5891.8125 - accuracy: 0.5775 -

```

precision: 0.5674 - recall: 0.6511 - auc: 0.5922 - val_loss: 4.7820 - val_tp:
1758.0000 - val_fp: 16652.0000 - val_tn: 11709.0000 - val_fn: 426.0000 -
val_accuracy: 0.4409 - val_precision: 0.0955 - val_recall: 0.8049 - val_auc:
0.6493
Epoch 14/100
31/31 [=====] - 1s 38ms/step - loss: 4.6762 - tp:
11058.5312 - fp: 8523.2812 - tn: 8421.6875 - fn: 5724.5000 - accuracy: 0.5785 -
precision: 0.5666 - recall: 0.6604 - auc: 0.5974 - val_loss: 4.5149 - val_tp:
1742.0000 - val_fp: 16046.0000 - val_tn: 12315.0000 - val_fn: 442.0000 -
val_accuracy: 0.4602 - val_precision: 0.0979 - val_recall: 0.7976 - val_auc:
0.6515
Epoch 15/100
31/31 [=====] - 1s 37ms/step - loss: 4.4203 - tp:
11088.4688 - fp: 8415.3438 - tn: 8483.5625 - fn: 5740.6250 - accuracy: 0.5785 -
precision: 0.5666 - recall: 0.6582 - auc: 0.5964 - val_loss: 4.2661 - val_tp:
1744.0000 - val_fp: 15906.0000 - val_tn: 12455.0000 - val_fn: 440.0000 -
val_accuracy: 0.4649 - val_precision: 0.0988 - val_recall: 0.7985 - val_auc:
0.6536
Epoch 16/100
31/31 [=====] - 1s 37ms/step - loss: 4.1801 - tp:
11032.6250 - fp: 8477.7812 - tn: 8570.1562 - fn: 5647.4375 - accuracy: 0.5814 -
precision: 0.5647 - recall: 0.6623 - auc: 0.5975 - val_loss: 4.0292 - val_tp:
1726.0000 - val_fp: 15618.0000 - val_tn: 12743.0000 - val_fn: 458.0000 -
val_accuracy: 0.4737 - val_precision: 0.0995 - val_recall: 0.7903 - val_auc:
0.6561
Epoch 17/100
31/31 [=====] - 1s 36ms/step - loss: 3.9448 - tp:
11346.4062 - fp: 8190.4688 - tn: 8566.3438 - fn: 5624.7812 - accuracy: 0.5914 -
precision: 0.5821 - recall: 0.6686 - auc: 0.6067 - val_loss: 3.8128 - val_tp:
1742.0000 - val_fp: 15553.0000 - val_tn: 12808.0000 - val_fn: 442.0000 -
val_accuracy: 0.4763 - val_precision: 0.1007 - val_recall: 0.7976 - val_auc:
0.6577
Epoch 18/100
31/31 [=====] - 1s 38ms/step - loss: 3.7238 - tp:
11382.2812 - fp: 8454.4062 - tn: 8479.5000 - fn: 5411.8125 - accuracy: 0.5891 -
precision: 0.5744 - recall: 0.6785 - auc: 0.6115 - val_loss: 3.6000 - val_tp:
1735.0000 - val_fp: 15356.0000 - val_tn: 13005.0000 - val_fn: 449.0000 -
val_accuracy: 0.4826 - val_precision: 0.1015 - val_recall: 0.7944 - val_auc:
0.6607
Epoch 19/100
31/31 [=====] - 1s 37ms/step - loss: 3.5194 - tp:
11524.9062 - fp: 8246.9688 - tn: 8480.8750 - fn: 5475.2500 - accuracy: 0.5944 -
precision: 0.5840 - recall: 0.6780 - auc: 0.6130 - val_loss: 3.4069 - val_tp:
1767.0000 - val_fp: 15477.0000 - val_tn: 12884.0000 - val_fn: 417.0000 -
val_accuracy: 0.4797 - val_precision: 0.1025 - val_recall: 0.8091 - val_auc:
0.6628
Epoch 20/100
31/31 [=====] - 1s 38ms/step - loss: 3.3259 - tp:

```

11737.2188 - fp: 8415.3750 - tn: 8334.0625 - fn: 5241.3438 - accuracy: 0.5944 -  
precision: 0.5823 - recall: 0.6907 - auc: 0.6134 - val\_loss: 3.2237 - val\_tp:  
1779.0000 - val\_fp: 15572.0000 - val\_tn: 12789.0000 - val\_fn: 405.0000 -  
val\_accuracy: 0.4769 - val\_precision: 0.1025 - val\_recall: 0.8146 - val\_auc:  
0.6642

Epoch 21/100

31/31 [=====] - 1s 38ms/step - loss: 3.1411 - tp:  
11614.4688 - fp: 8401.1875 - tn: 8527.9688 - fn: 5184.3750 - accuracy: 0.5966 -  
precision: 0.5799 - recall: 0.6907 - auc: 0.6185 - val\_loss: 3.0446 - val\_tp:  
1772.0000 - val\_fp: 15381.0000 - val\_tn: 12980.0000 - val\_fn: 412.0000 -  
val\_accuracy: 0.4830 - val\_precision: 0.1033 - val\_recall: 0.8114 - val\_auc:  
0.6655

Epoch 22/100

31/31 [=====] - 1s 38ms/step - loss: 2.9675 - tp:  
11588.3750 - fp: 8440.6250 - tn: 8544.7812 - fn: 5154.2188 - accuracy: 0.5958 -  
precision: 0.5759 - recall: 0.6920 - auc: 0.6200 - val\_loss: 2.8771 - val\_tp:  
1778.0000 - val\_fp: 15338.0000 - val\_tn: 13023.0000 - val\_fn: 406.0000 -  
val\_accuracy: 0.4846 - val\_precision: 0.1039 - val\_recall: 0.8141 - val\_auc:  
0.6672

Epoch 23/100

31/31 [=====] - 1s 38ms/step - loss: 2.8008 - tp:  
11900.9375 - fp: 8233.2188 - tn: 8518.7812 - fn: 5075.0625 - accuracy: 0.6054 -  
precision: 0.5913 - recall: 0.6992 - auc: 0.6282 - val\_loss: 2.7263 - val\_tp:  
1792.0000 - val\_fp: 15345.0000 - val\_tn: 13016.0000 - val\_fn: 392.0000 -  
val\_accuracy: 0.4848 - val\_precision: 0.1046 - val\_recall: 0.8205 - val\_auc:  
0.6691

Epoch 24/100

31/31 [=====] - 1s 37ms/step - loss: 2.6514 - tp:  
11854.5625 - fp: 8408.8125 - tn: 8502.1875 - fn: 4962.4375 - accuracy: 0.6027 -  
precision: 0.5831 - recall: 0.7044 - auc: 0.6259 - val\_loss: 2.5828 - val\_tp:  
1793.0000 - val\_fp: 15322.0000 - val\_tn: 13039.0000 - val\_fn: 391.0000 -  
val\_accuracy: 0.4856 - val\_precision: 0.1048 - val\_recall: 0.8210 - val\_auc:  
0.6703

Epoch 25/100

31/31 [=====] - 1s 37ms/step - loss: 2.5050 - tp:  
12016.2188 - fp: 8380.8750 - tn: 8508.2500 - fn: 4822.6562 - accuracy: 0.6082 -  
precision: 0.5897 - recall: 0.7123 - auc: 0.6310 - val\_loss: 2.4444 - val\_tp:  
1800.0000 - val\_fp: 15484.0000 - val\_tn: 12877.0000 - val\_fn: 384.0000 -  
val\_accuracy: 0.4805 - val\_precision: 0.1041 - val\_recall: 0.8242 - val\_auc:  
0.6712

Epoch 26/100

31/31 [=====] - 1s 38ms/step - loss: 2.3666 - tp:  
12083.2188 - fp: 8370.4688 - tn: 8535.8438 - fn: 4738.4688 - accuracy: 0.6116 -  
precision: 0.5900 - recall: 0.7194 - auc: 0.6368 - val\_loss: 2.3160 - val\_tp:  
1798.0000 - val\_fp: 15489.0000 - val\_tn: 12872.0000 - val\_fn: 386.0000 -  
val\_accuracy: 0.4803 - val\_precision: 0.1040 - val\_recall: 0.8233 - val\_auc:  
0.6726

Epoch 27/100



31/31 [=====] - 1s 38ms/step - loss: 2.2401 - tp: 12169.6875 - fp: 8367.4062 - tn: 8530.8438 - fn: 4660.0625 - accuracy: 0.6133 - precision: 0.5916 - recall: 0.7232 - auc: 0.6387 - val\_loss: 2.1945 - val\_tp: 1796.0000 - val\_fp: 15467.0000 - val\_tn: 12894.0000 - val\_fn: 388.0000 - val\_accuracy: 0.4809 - val\_precision: 0.1040 - val\_recall: 0.8223 - val\_auc: 0.6716

Epoch 28/100

31/31 [=====] - 1s 40ms/step - loss: 2.1237 - tp: 12277.0625 - fp: 8243.9688 - tn: 8549.2812 - fn: 4657.6875 - accuracy: 0.6168 - precision: 0.5976 - recall: 0.7233 - auc: 0.6414 - val\_loss: 2.0850 - val\_tp: 1784.0000 - val\_fp: 15522.0000 - val\_tn: 12839.0000 - val\_fn: 400.0000 - val\_accuracy: 0.4787 - val\_precision: 0.1031 - val\_recall: 0.8168 - val\_auc: 0.6733

Epoch 29/100

31/31 [=====] - 1s 37ms/step - loss: 2.0108 - tp: 12308.7188 - fp: 8331.0625 - tn: 8587.5000 - fn: 4500.7188 - accuracy: 0.6200 - precision: 0.5958 - recall: 0.7336 - auc: 0.6490 - val\_loss: 1.9801 - val\_tp: 1793.0000 - val\_fp: 15505.0000 - val\_tn: 12856.0000 - val\_fn: 391.0000 - val\_accuracy: 0.4796 - val\_precision: 0.1037 - val\_recall: 0.8210 - val\_auc: 0.6731

Epoch 30/100

31/31 [=====] - 1s 38ms/step - loss: 1.9082 - tp: 12403.9375 - fp: 8299.5312 - tn: 8612.2500 - fn: 4412.2812 - accuracy: 0.6238 - precision: 0.5992 - recall: 0.7376 - auc: 0.6499 - val\_loss: 1.8829 - val\_tp: 1808.0000 - val\_fp: 15688.0000 - val\_tn: 12673.0000 - val\_fn: 376.0000 - val\_accuracy: 0.4741 - val\_precision: 0.1033 - val\_recall: 0.8278 - val\_auc: 0.6753

Epoch 31/100

31/31 [=====] - 1s 38ms/step - loss: 1.8153 - tp: 12489.9688 - fp: 8342.9062 - tn: 8521.3750 - fn: 4373.7500 - accuracy: 0.6222 - precision: 0.5985 - recall: 0.7413 - auc: 0.6481 - val\_loss: 1.7952 - val\_tp: 1865.0000 - val\_fp: 15885.0000 - val\_tn: 12476.0000 - val\_fn: 319.0000 - val\_accuracy: 0.4695 - val\_precision: 0.1051 - val\_recall: 0.8539 - val\_auc: 0.6772

Epoch 32/100

31/31 [=====] - 1s 39ms/step - loss: 1.7230 - tp: 12503.0312 - fp: 8347.6875 - tn: 8561.8750 - fn: 4315.4062 - accuracy: 0.6250 - precision: 0.6002 - recall: 0.7425 - auc: 0.6581 - val\_loss: 1.7075 - val\_tp: 1853.0000 - val\_fp: 15909.0000 - val\_tn: 12452.0000 - val\_fn: 331.0000 - val\_accuracy: 0.4683 - val\_precision: 0.1043 - val\_recall: 0.8484 - val\_auc: 0.6801

Epoch 33/100

31/31 [=====] - 1s 37ms/step - loss: 1.6419 - tp: 12683.1250 - fp: 8210.7500 - tn: 8573.6250 - fn: 4260.5000 - accuracy: 0.6299 - precision: 0.6079 - recall: 0.7476 - auc: 0.6571 - val\_loss: 1.6311 - val\_tp: 1885.0000 - val\_fp: 16218.0000 - val\_tn: 12143.0000 - val\_fn: 299.0000 - val\_accuracy: 0.4593 - val\_precision: 0.1041 - val\_recall: 0.8631 - val\_auc: 0.6836

Epoch 34/100

31/31 [=====] - 1s 38ms/step - loss: 1.5641 - tp: 12862.7188 - fp: 8276.0000 - tn: 8442.6875 - fn: 4146.5938 - accuracy: 0.6309 - precision: 0.6087 - recall: 0.7547 - auc: 0.6608 - val\_loss: 1.5623 - val\_tp: 1890.0000 - val\_fp: 16285.0000 - val\_tn: 12076.0000 - val\_fn: 294.0000 - val\_accuracy: 0.4572 - val\_precision: 0.1040 - val\_recall: 0.8654 - val\_auc: 0.6857

Epoch 35/100

31/31 [=====] - 1s 39ms/step - loss: 1.4907 - tp: 12694.9062 - fp: 8422.8438 - tn: 8576.1875 - fn: 4034.0625 - accuracy: 0.6306 - precision: 0.5998 - recall: 0.7589 - auc: 0.6626 - val\_loss: 1.4937 - val\_tp: 1908.0000 - val\_fp: 16472.0000 - val\_tn: 11889.0000 - val\_fn: 276.0000 - val\_accuracy: 0.4517 - val\_precision: 0.1038 - val\_recall: 0.8736 - val\_auc: 0.6848

Epoch 36/100

31/31 [=====] - 1s 39ms/step - loss: 1.4206 - tp: 12856.5312 - fp: 8243.3750 - tn: 8650.7812 - fn: 3977.3125 - accuracy: 0.6383 - precision: 0.6099 - recall: 0.7642 - auc: 0.6708 - val\_loss: 1.4339 - val\_tp: 1913.0000 - val\_fp: 16486.0000 - val\_tn: 11875.0000 - val\_fn: 271.0000 - val\_accuracy: 0.4514 - val\_precision: 0.1040 - val\_recall: 0.8759 - val\_auc: 0.6864

Epoch 37/100

31/31 [=====] - 1s 38ms/step - loss: 1.3564 - tp: 12869.4688 - fp: 8217.2188 - tn: 8684.5938 - fn: 3956.7188 - accuracy: 0.6380 - precision: 0.6085 - recall: 0.7640 - auc: 0.6734 - val\_loss: 1.3816 - val\_tp: 1962.0000 - val\_fp: 17228.0000 - val\_tn: 11133.0000 - val\_fn: 222.0000 - val\_accuracy: 0.4287 - val\_precision: 0.1022 - val\_recall: 0.8984 - val\_auc: 0.6866

Epoch 38/100

31/31 [=====] - 1s 39ms/step - loss: 1.2952 - tp: 13093.5938 - fp: 8252.8438 - tn: 8612.6562 - fn: 3768.9062 - accuracy: 0.6441 - precision: 0.6144 - recall: 0.7769 - auc: 0.6771 - val\_loss: 1.3356 - val\_tp: 1956.0000 - val\_fp: 17250.0000 - val\_tn: 11111.0000 - val\_fn: 228.0000 - val\_accuracy: 0.4278 - val\_precision: 0.1018 - val\_recall: 0.8956 - val\_auc: 0.6885

Epoch 39/100

31/31 [=====] - 1s 39ms/step - loss: 1.2409 - tp: 13298.6562 - fp: 8152.7812 - tn: 8556.3438 - fn: 3720.2188 - accuracy: 0.6488 - precision: 0.6206 - recall: 0.7813 - auc: 0.6821 - val\_loss: 1.2949 - val\_tp: 2011.0000 - val\_fp: 18243.0000 - val\_tn: 10118.0000 - val\_fn: 173.0000 - val\_accuracy: 0.3971 - val\_precision: 0.0993 - val\_recall: 0.9208 - val\_auc: 0.6909

Epoch 40/100

31/31 [=====] - 1s 40ms/step - loss: 1.1943 - tp: 13250.2500 - fp: 8439.0938 - tn: 8461.0000 - fn: 3577.6562 - accuracy: 0.6420 - precision: 0.6093 - recall: 0.7864 - auc: 0.6769 - val\_loss: 1.2531 - val\_tp: 1999.0000 - val\_fp: 17979.0000 - val\_tn: 10382.0000 - val\_fn: 185.0000 - val\_accuracy: 0.4053 - val\_precision: 0.1001 - val\_recall: 0.9153 - val\_auc:

0.6947

Epoch 41/100

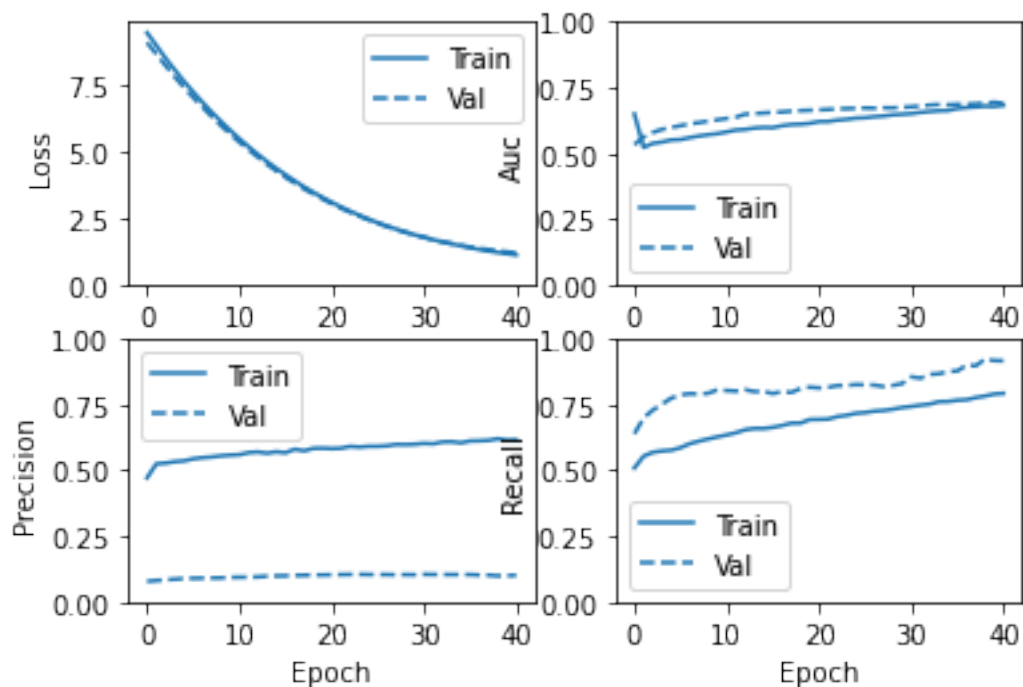
```
31/31 [=====] - 1s 38ms/step - loss: 1.1453 - tp:
13321.3438 - fp: 8354.4375 - tn: 8516.0938 - fn: 3536.1250 - accuracy: 0.6472 -
precision: 0.6156 - recall: 0.7892 - auc: 0.6841 - val_loss: 1.2138 - val_tp:
1997.0000 - val_fp: 17768.0000 - val_tn: 10593.0000 - val_fn: 187.0000 -
val_accuracy: 0.4122 - val_precision: 0.1010 - val_recall: 0.9144 - val_auc:
0.6903
```

Restoring model weights from the end of the best epoch.

Epoch 00041: early stopping

```
[ ]: plot_metrics(resampled_history)
```

```
# This model seems to have performed the best by far
```



```
[ ]: !mkdir -p saved_model
resampled_three_layer_model.save('saved_model/resampled_three_layer_model')
```

```
INFO:tensorflow:Assets written to:
saved_model/resampled_three_layer_model/assets
```

```
[ ]: train_predictions_resampled = resampled_three_layer_model.
    →predict(train_features, batch_size=BATCH_SIZE)
test_predictions_resampled = resampled_three_layer_model.predict(test_features,
    →batch_size=BATCH_SIZE)
```

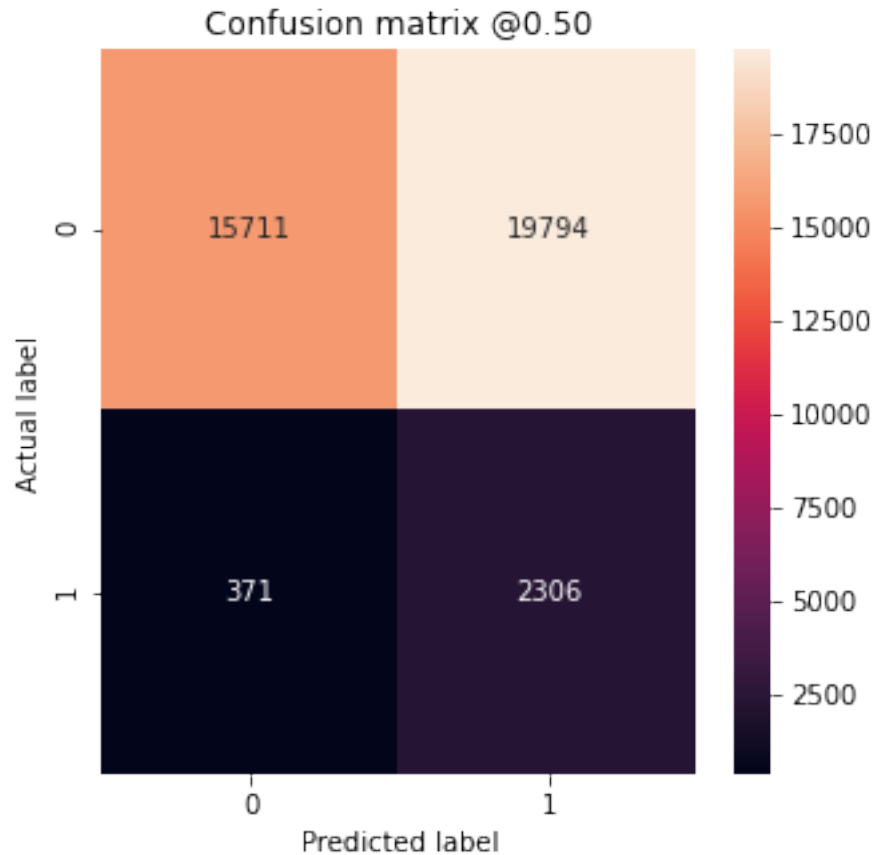
```
[ ]: resampled_results = resampled_three_layer_model.evaluate(test_features,
    ↪test_labels,
                                batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(resampled_three_layer_model.metrics_names,
    ↪resampled_results):
    print(name, ': ', value)
print()

plot_cm(test_labels, test_predictions_resampled)
```

```
loss : 1.7949222326278687
tp : 2306.0
fp : 19794.0
tn : 15711.0
fn : 371.0
accuracy : 0.471871554851532
precision : 0.10434389114379883
recall : 0.8614120483398438
auc : 0.6911264061927795
```

```
f1 Score 0.1861403721193042
```

```
Irrelevant Documents Detected (True Negatives): 15711
Irrelevant Documents Incorrectly Detected (False Positives): 19794
Relevant Documents Missed (False Negatives): 371
Relevant Documents Detected (True Positives): 2306
Total Relevant Documents: 2677
```



```
[ ]: pred_labels = resampled_three_layer_model.predict(test_features)
pred_labels = pred_labels.round(0)
pred_labels = pred_labels.astype(int)

tl_resampled = {'Model': 'Resampled Three Layer Model',
                'Loss': resampled_results[0],
                'Accuracy': accuracy_score(test_labels, pred_labels),
                'Precision': precision_score(test_labels, pred_labels),
                'Recall': recall_score(test_labels, pred_labels),
                'F1 Score': f1_score(test_labels, pred_labels)}

tl_resampled
```

```
[ ]: {'Accuracy': 0.47187156251636897,
      'F1 Score': 0.1861403721193042,
      'Loss': 1.7949222326278687,
      'Model': 'Resampled Three Layer Model',
      'Precision': 0.10434389140271494,
      'Recall': 0.8614120283899888}
```

```
[ ]: test_predictions_baseline = resampled_three_layer_model.predict(test)
```

```
[ ]: test_predictions_baseline = test_predictions_baseline.round(0)
test_predictions_baseline = test_predictions_baseline.astype(int)
test_predictions_baseline
```

```
[ ]: array([[1],
          [0],
          [1],
          ...,
          [0],
          [0],
          [0]])
```

```
[ ]: predictions_baseline= pd.DataFrame(test_predictions_baseline)
predictions_baseline['Id'] = predictions_baseline.index
predictions_baseline.rename(columns={ predictions_baseline.columns[0]: "psrel"
→}, inplace = True)
predictions_baseline = predictions_baseline[['Id', 'psrel']]
predictions_baseline
```

```
[ ]:      Id  psrel
0      0      1
1      1      0
2      2      1
3      3      1
4      4      1
...    ...    ...
4995  4995      1
4996  4996      1
4997  4997      0
4998  4998      0
4999  4999      0
```

[5000 rows x 2 columns]

```
[ ]: predictions_baseline['psrel'].value_counts()
```

```
[ ]: 0    2663
1    2337
Name: psrel, dtype: int64
```

```
[ ]: predictions_baseline.to_csv('Resampled colab predictions.csv', index=False)
```

```
[ ]: test_predictions_baseline = resampled_three_layer_model.predict(test_features,
→batch_size=BATCH_SIZE)
```

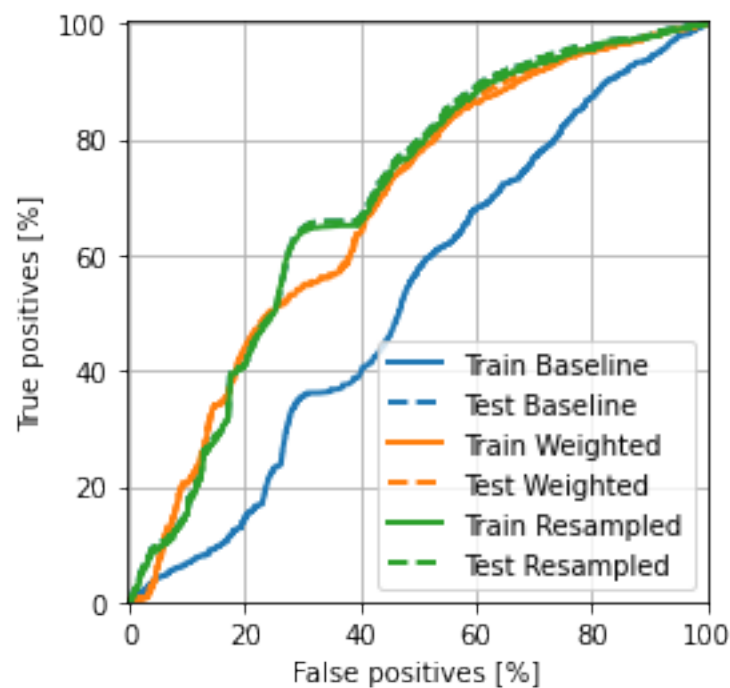
## Final Evaluation

```
[ ]: plot_roc("Train Baseline", train_labels, train_predictions_baseline,
    →color=colors[0])
plot_roc("Test Baseline", test_labels, test_predictions_baseline,
    →color=colors[0], linestyle='--')

plot_roc("Train Weighted", train_labels, train_predictions_weighted,
    →color=colors[1])
plot_roc("Test Weighted", test_labels, test_predictions_weighted,
    →color=colors[1], linestyle='--')

plot_roc("Train Resampled", train_labels, train_predictions_resampled,
    →color=colors[2])
plot_roc("Test Resampled", test_labels, test_predictions_resampled,
    →color=colors[2], linestyle='--')
plt.legend(loc='lower right')
```

```
[ ]: <matplotlib.legend.Legend at 0x7f7509537d50>
```



```
[ ]: keras.backend.clear_session()
```

### Predictions

```
[ ]: weighted_three_layer_model = tf.keras.models.load_model('saved_model/
    →weighted_three_layer_model')
```

```
[ ]: test_predictions = weighted_three_layer_model.predict(test)
```

```
[ ]: test_predictions = test_predictions.round(0)
test_predictions = test_predictions.astype(int)
test_predictions
```

```
[ ]: array([[0],
          [0],
          [0],
          ...,
          [0],
          [0],
          [0]])
```

```
[ ]: predictions_baseline= pd.DataFrame(test_predictions)
predictions_baseline['Id'] = predictions_baseline.index
predictions_baseline.rename(columns={ predictions_baseline.columns[0]: "psrel"↵
→}, inplace = True)
predictions_baseline = predictions_baseline[['Id','psrel']]
```

```
[ ]: predictions_baseline['psrel'].value_counts()
```

```
[ ]: 0    4711
     1     289
     Name: psrel, dtype: int64
```

```
[ ]: # create output file
predictions_baseline.to_csv('three-layer predictions.csv', index=False)
```

### 6.2.2 Random Forest with Updated Class Weights

```
[ ]: wt_rnd_clf = RandomForestClassifier(criterion = "gini",
                                       n_estimators=200,
                                       max_leaf_nodes=100,
                                       min_samples_leaf=1,
                                       max_features="sqrt",
                                       class_weight=class_weight,
                                       n_jobs=-1)
```

```
[ ]: wt_rnd_clf.fit(train_features, train_labels)

pred_labels = wt_rnd_clf.predict(test_features)

scores = cross_validate(wt_rnd_clf, train_features, train_labels,
                        scoring=("accuracy","precision","recall","f1"),cv=10)

wt_rnd = {'Model': 'Weighted Random Forest',
```



```
'Loss': 0,  
'Accuracy': accuracy_score(test_labels, pred_labels),  
'Precision': precision_score(test_labels, pred_labels),  
'Recall': recall_score(test_labels, pred_labels),  
'F1 Score': f1_score(test_labels, pred_labels)}
```

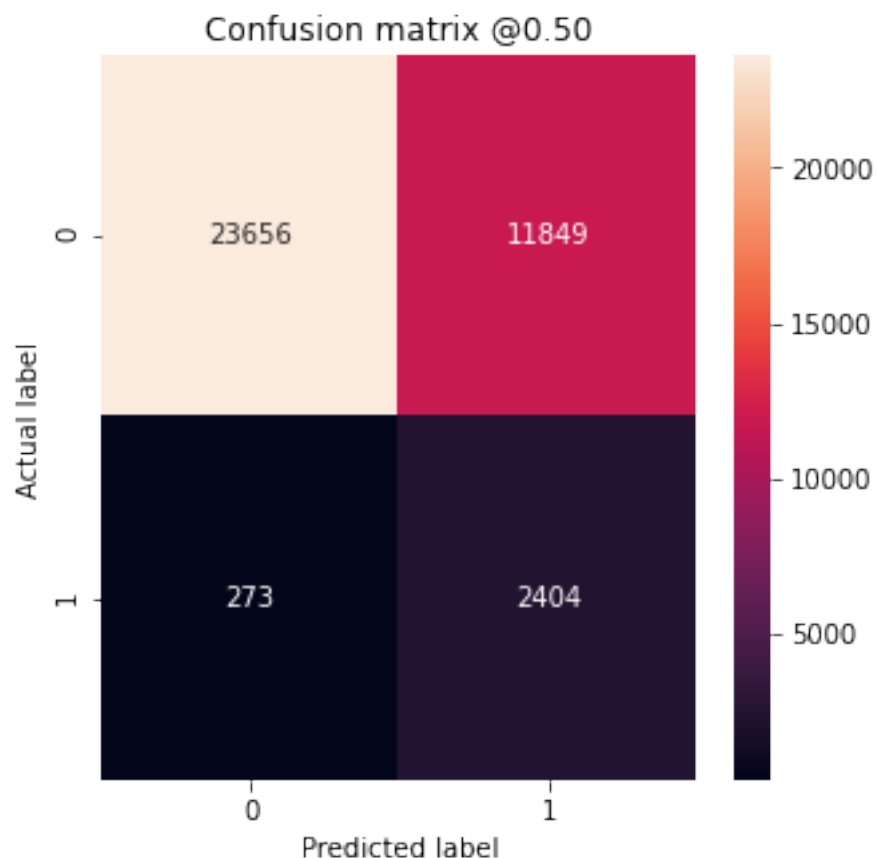
```
wt_rnd
```

```
[ ]: {'Accuracy': 0.6825205594259075,  
      'F1 Score': 0.2839929119905493,  
      'Loss': 0,  
      'Model': 'Weighted Random Forest',  
      'Precision': 0.1686662457026591,  
      'Recall': 0.8980201718341427}
```

```
[ ]: train_predictions_wt_rnd_clf = wt_rnd_clf.predict(train_features)  
      test_predictions_wt_rnd_clf = wt_rnd_clf.predict(test_features)  
  
      plot_cm(test_labels, test_predictions_wt_rnd_clf)
```

```
f1 Score 0.2839929119905493
```

```
Irrelevant Documents Detected (True Negatives): 23656  
Irrelevant Documents Incorrectly Detected (False Positives): 11849  
Relevant Documents Missed (False Negatives): 273  
Relevant Documents Detected (True Positives): 2404  
Total Relevant Documents: 2677
```



```
[ ]: test_predictions_wt_rnd_clf = wt_rnd_clf.predict(test)
test_predictions_wt_rnd_clf = test_predictions_wt_rnd_clf.round(0)
test_predictions_wt_rnd_clf = test_predictions_wt_rnd_clf.astype(int)
test_predictions_wt_rnd_clf
```

```
[ ]: array([0, 0, 0, ..., 0, 0, 0])
```

```
[ ]: predictions_wt_rnd_clf= pd.DataFrame(test_predictions_wt_rnd_clf)
predictions_wt_rnd_clf['Id'] = predictions_wt_rnd_clf.index
predictions_wt_rnd_clf.rename(columns={ predictions_wt_rnd_clf.columns[0]:  
    ↳"psrel" }, inplace = True)
predictions_wt_rnd_clf = predictions_wt_rnd_clf[['Id', 'psrel']]
predictions_wt_rnd_clf
```

```
[ ]:      Id  psrel
0      0      0
1      1      0
2      2      0
3      3      0
```

```

4          4          0
...      ...      ...
4995  4995      0
4996  4996      0
4997  4997      0
4998  4998      0
4999  4999      0

```

[5000 rows x 2 columns]

```
[ ]: predictions_wt_rnd_clf['psrel'].value_counts()
```

```
[ ]: 0    4198
      1     802
      Name: psrel, dtype: int64
```

```
[ ]: predictions_wt_rnd_clf.to_csv('Reweighted Random Forest colab predictions.csv',
    ↪index=False)
```

Baseline Model also performs much better with updated weights

### 6.2.3 Deep Network Model

#### Dense Deep Model Function

```
[ ]: early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_precision',
    verbose=1,
    patience=10,
    mode='max',
    restore_best_weights=True)
```

```
[ ]: def build_dense_deep(n_hidden=5, n_neurons=30, learning_rate=3e-4,
    ↪metrics=METRICS, output_bias=None, input_shape=train_features.shape[1:]):
    if output_bias is not None:
        output_bias = tf.keras.initializers.Constant(output_bias)
    model = keras.models.Sequential()
    model.add(keras.layers.InputLayer(input_shape=input_shape))
    for layer in range(n_hidden):
        model.add(RegularizedDense(n_neurons))
        model.add(keras.layers.BatchNormalization())
        model.add(keras.layers.Dropout(rate=0.3))
    model.add(keras.layers.Dense(1, activation="sigmoid",
    ↪bias_initializer=output_bias))

    optimizer = keras.optimizers.Nadam(lr=learning_rate)
    model.compile(loss="binary_crossentropy", optimizer=optimizer, metrics=metrics)
    return model
```

```
[ ]: # Create a default dense deep model using the build_dense_deep function
```

```
dense_deep_keras_reg = keras.wrappers.scikit_learn.  
    ↳KerasRegressor(build_dense_deep)  
  
dense_deep_model = build_dense_deep()  
dense_deep_model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 30)	2250
batch_normalization (Batch Normalization)	(None, 30)	120
dropout (Dropout)	(None, 30)	0
dense_1 (Dense)	(None, 30)	930
batch_normalization_1 (Batch Normalization)	(None, 30)	120
dropout_1 (Dropout)	(None, 30)	0
dense_2 (Dense)	(None, 30)	930
batch_normalization_2 (Batch Normalization)	(None, 30)	120
dropout_2 (Dropout)	(None, 30)	0
dense_3 (Dense)	(None, 30)	930
batch_normalization_3 (Batch Normalization)	(None, 30)	120
dropout_3 (Dropout)	(None, 30)	0
dense_4 (Dense)	(None, 30)	930
batch_normalization_4 (Batch Normalization)	(None, 30)	120
dropout_4 (Dropout)	(None, 30)	0
dense_5 (Dense)	(None, 1)	31

Total params: 6,601  
 Trainable params: 6,301  
 Non-trainable params: 300

## With Class Weights

```
[ ]: # Find initial weights
initial_weights = os.path.join(tempfile.mkdtemp(), 'initial_weights')
dense_deep_model.save_weights(initial_weights)
```

```
[ ]: # Load in previously calculated weights

dense_deep_model.load_weights(initial_weights)

dense_deep_history = dense_deep_model.fit(
    train_features,
    train_labels,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    callbacks=[early_stopping],
    validation_data=(val_features, val_labels),
    class_weight=class_weight)
```

Epoch 1/100

60/60 [=====] - 8s 47ms/step - loss: 14.6585 - tp: 3616.5246 - fp: 35217.3115 - tn: 59074.1803 - fn: 3705.4262 - accuracy: 0.6030 - precision: 0.0941 - recall: 0.5294 - auc: 0.5715 - val\_loss: 13.3116 - val\_tp: 1271.0000 - val\_fp: 10698.0000 - val\_tn: 17663.0000 - val\_fn: 913.0000 - val\_accuracy: 0.6199 - val\_precision: 0.1062 - val\_recall: 0.5820 - val\_auc: 0.6492

Epoch 2/100

60/60 [=====] - 2s 29ms/step - loss: 13.4353 - tp: 1630.4590 - fp: 16644.7541 - tn: 42142.6557 - fn: 3013.5738 - accuracy: 0.6911 - precision: 0.0886 - recall: 0.3431 - auc: 0.5612 - val\_loss: 12.1671 - val\_tp: 1252.0000 - val\_fp: 9969.0000 - val\_tn: 18392.0000 - val\_fn: 932.0000 - val\_accuracy: 0.6431 - val\_precision: 0.1116 - val\_recall: 0.5733 - val\_auc: 0.6655

Epoch 3/100

60/60 [=====] - 2s 28ms/step - loss: 12.2787 - tp: 1793.7213 - fp: 18013.9180 - tn: 40797.0656 - fn: 2826.7377 - accuracy: 0.6726 - precision: 0.0905 - recall: 0.3852 - auc: 0.5681 - val\_loss: 11.1066 - val\_tp: 1234.0000 - val\_fp: 9766.0000 - val\_tn: 18595.0000 - val\_fn: 950.0000 - val\_accuracy: 0.6492 - val\_precision: 0.1122 - val\_recall: 0.5650 - val\_auc: 0.6729

Epoch 4/100

60/60 [=====] - 2s 29ms/step - loss: 11.1970 - tp: 1974.0984 - fp: 19124.2787 - tn: 39676.0656 - fn: 2657.0000 - accuracy: 0.6576 - precision: 0.0947 - recall: 0.4282 - auc: 0.5814 - val\_loss: 10.1292 - val\_tp: 1236.0000 - val\_fp: 9803.0000 - val\_tn: 18558.0000 - val\_fn: 948.0000 - val\_accuracy: 0.6480 - val\_precision: 0.1120 - val\_recall: 0.5659 - val\_auc: 0.6799

Epoch 5/100

60/60 [=====] - 2s 28ms/step - loss: 10.1948 - tp:

2121.7869 - fp: 20261.2459 - tn: 38539.9180 - fn: 2508.4918 - accuracy: 0.6417 -  
precision: 0.0945 - recall: 0.4594 - auc: 0.5883 - val\_loss: 9.2339 - val\_tp:  
1289.0000 - val\_fp: 10017.0000 - val\_tn: 18344.0000 - val\_fn: 895.0000 -  
val\_accuracy: 0.6428 - val\_precision: 0.1140 - val\_recall: 0.5902 - val\_auc:  
0.6829

Epoch 6/100

60/60 [=====] - 2s 28ms/step - loss: 9.2845 - tp:  
2197.9180 - fp: 21408.1967 - tn: 37432.3115 - fn: 2393.0164 - accuracy: 0.6276 -  
precision: 0.0943 - recall: 0.4811 - auc: 0.5894 - val\_loss: 8.3992 - val\_tp:  
1328.0000 - val\_fp: 10256.0000 - val\_tn: 18105.0000 - val\_fn: 856.0000 -  
val\_accuracy: 0.6362 - val\_precision: 0.1146 - val\_recall: 0.6081 - val\_auc:  
0.6873

Epoch 7/100

60/60 [=====] - 2s 29ms/step - loss: 8.4356 - tp:  
2361.4590 - fp: 22568.6230 - tn: 36268.6721 - fn: 2232.6885 - accuracy: 0.6100 -  
precision: 0.0942 - recall: 0.5105 - auc: 0.5922 - val\_loss: 7.6385 - val\_tp:  
1356.0000 - val\_fp: 10743.0000 - val\_tn: 17618.0000 - val\_fn: 828.0000 -  
val\_accuracy: 0.6212 - val\_precision: 0.1121 - val\_recall: 0.6209 - val\_auc:  
0.6845

Epoch 8/100

60/60 [=====] - 2s 29ms/step - loss: 7.6606 - tp:  
2475.4754 - fp: 23691.3443 - tn: 35152.9836 - fn: 2111.6393 - accuracy: 0.5942 -  
precision: 0.0944 - recall: 0.5386 - auc: 0.5975 - val\_loss: 6.9359 - val\_tp:  
1408.0000 - val\_fp: 11220.0000 - val\_tn: 17141.0000 - val\_fn: 776.0000 -  
val\_accuracy: 0.6073 - val\_precision: 0.1115 - val\_recall: 0.6447 - val\_auc:  
0.6903

Epoch 9/100

60/60 [=====] - 2s 28ms/step - loss: 6.9635 - tp:  
2531.4754 - fp: 24526.3770 - tn: 34286.1639 - fn: 2087.4262 - accuracy: 0.5813 -  
precision: 0.0940 - recall: 0.5461 - auc: 0.5927 - val\_loss: 6.2910 - val\_tp:  
1478.0000 - val\_fp: 11809.0000 - val\_tn: 16552.0000 - val\_fn: 706.0000 -  
val\_accuracy: 0.5903 - val\_precision: 0.1112 - val\_recall: 0.6767 - val\_auc:  
0.6852

Epoch 10/100

60/60 [=====] - 2s 28ms/step - loss: 6.3074 - tp:  
2640.5410 - fp: 25753.3770 - tn: 33056.2131 - fn: 1981.3115 - accuracy: 0.5635 -  
precision: 0.0940 - recall: 0.5729 - auc: 0.5936 - val\_loss: 5.6959 - val\_tp:  
1553.0000 - val\_fp: 12209.0000 - val\_tn: 16152.0000 - val\_fn: 631.0000 -  
val\_accuracy: 0.5796 - val\_precision: 0.1128 - val\_recall: 0.7111 - val\_auc:  
0.6908

Epoch 11/100

60/60 [=====] - 2s 29ms/step - loss: 5.6998 - tp:  
2730.7705 - fp: 26445.6230 - tn: 32406.8361 - fn: 1848.2131 - accuracy: 0.5555 -  
precision: 0.0936 - recall: 0.5956 - auc: 0.5981 - val\_loss: 5.1588 - val\_tp:  
1609.0000 - val\_fp: 12947.0000 - val\_tn: 15414.0000 - val\_fn: 575.0000 -  
val\_accuracy: 0.5573 - val\_precision: 0.1105 - val\_recall: 0.7367 - val\_auc:  
0.6889

Epoch 12/100

```
60/60 [=====] - 2s 28ms/step - loss: 5.1496 - tp:
2847.0328 - fp: 27379.5246 - tn: 31464.2951 - fn: 1740.5902 - accuracy: 0.5409 -
precision: 0.0944 - recall: 0.6225 - auc: 0.6066 - val_loss: 4.6687 - val_tp:
1669.0000 - val_fp: 13723.0000 - val_tn: 14638.0000 - val_fn: 515.0000 -
val_accuracy: 0.5339 - val_precision: 0.1084 - val_recall: 0.7642 - val_auc:
0.6895
```

Epoch 13/100

```
60/60 [=====] - 2s 28ms/step - loss: 4.6602 - tp:
2966.0492 - fp: 28095.0328 - tn: 30691.3443 - fn: 1679.0164 - accuracy: 0.5319 -
precision: 0.0959 - recall: 0.6400 - auc: 0.6078 - val_loss: 4.2259 - val_tp:
1721.0000 - val_fp: 14163.0000 - val_tn: 14198.0000 - val_fn: 463.0000 -
val_accuracy: 0.5212 - val_precision: 0.1083 - val_recall: 0.7880 - val_auc:
0.6835
```

Epoch 14/100

```
60/60 [=====] - 2s 30ms/step - loss: 4.2143 - tp:
2948.1475 - fp: 28530.0656 - tn: 30310.3443 - fn: 1642.8852 - accuracy: 0.5251 -
precision: 0.0941 - recall: 0.6418 - auc: 0.6038 - val_loss: 3.8231 - val_tp:
1769.0000 - val_fp: 14811.0000 - val_tn: 13550.0000 - val_fn: 415.0000 -
val_accuracy: 0.5015 - val_precision: 0.1067 - val_recall: 0.8100 - val_auc:
0.6827
```

Epoch 15/100

```
60/60 [=====] - 2s 28ms/step - loss: 3.8034 - tp:
3065.7705 - fp: 28955.5738 - tn: 29842.8689 - fn: 1567.2295 - accuracy: 0.5202 -
precision: 0.0956 - recall: 0.6595 - auc: 0.6091 - val_loss: 3.4566 - val_tp:
1780.0000 - val_fp: 15138.0000 - val_tn: 13223.0000 - val_fn: 404.0000 -
val_accuracy: 0.4912 - val_precision: 0.1052 - val_recall: 0.8150 - val_auc:
0.6809
```

Epoch 16/100

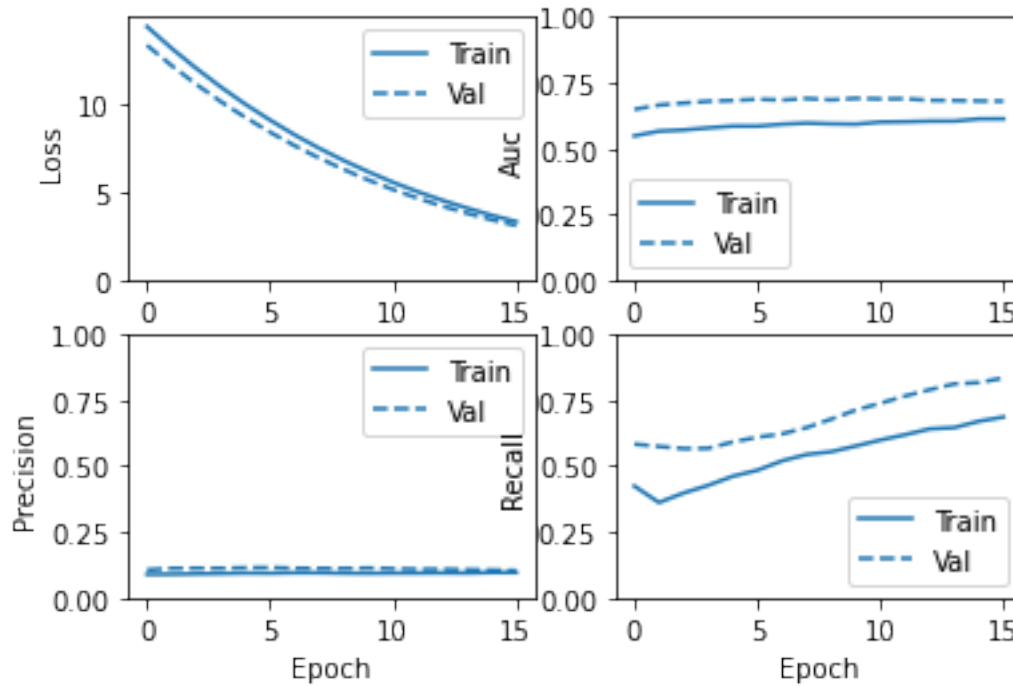
```
60/60 [=====] - 2s 28ms/step - loss: 3.4391 - tp:
3141.0820 - fp: 29640.2295 - tn: 29164.3770 - fn: 1485.7541 - accuracy: 0.5095 -
precision: 0.0960 - recall: 0.6766 - auc: 0.6098 - val_loss: 3.1283 - val_tp:
1818.0000 - val_fp: 15753.0000 - val_tn: 12608.0000 - val_fn: 366.0000 -
val_accuracy: 0.4723 - val_precision: 0.1035 - val_recall: 0.8324 - val_auc:
0.6802
```

Restoring model weights from the end of the best epoch.

Epoch 00016: early stopping

## Evaluation

```
[ ]: plot_metrics(dense_deep_history)
```



```
[ ]: train_predictions_dense_deep_model = dense_deep_model.predict(train_features,
    ↳batch_size=BATCH_SIZE)
test_predictions_dense_deep_model = dense_deep_model.predict(test_features,
    ↳batch_size=BATCH_SIZE)
```

```
[ ]: dense_deep_results = dense_deep_model.evaluate(test_features, test_labels,
    batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(dense_deep_model.metrics_names, dense_deep_results):
    print(name, ': ', value)
print()

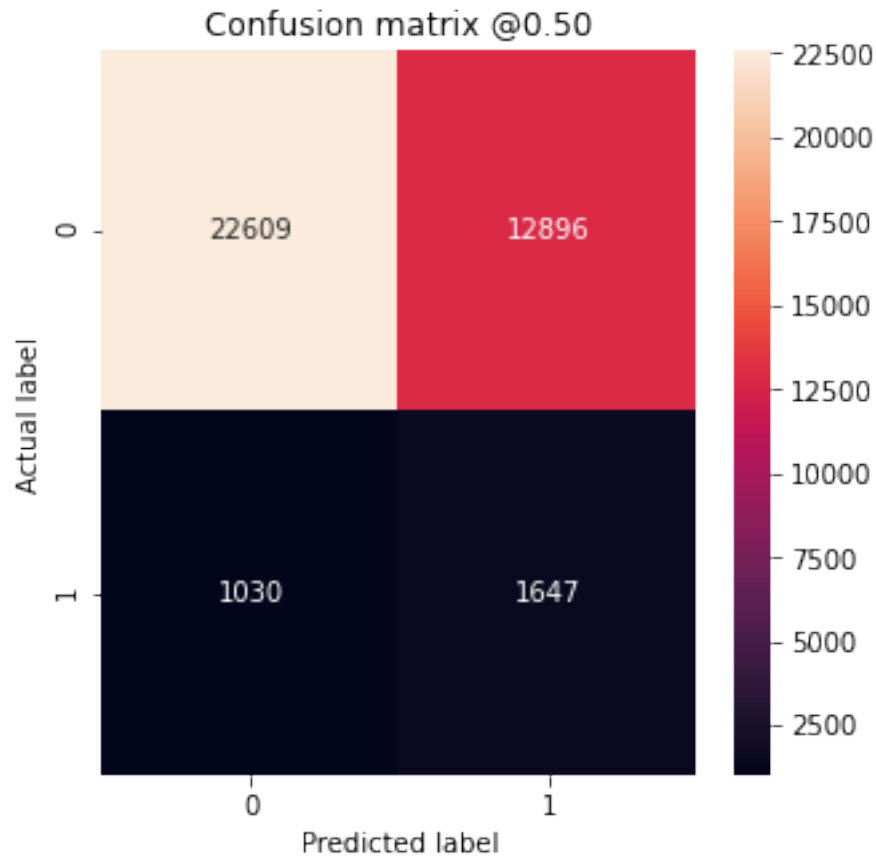
plot_cm(test_labels, test_predictions_dense_deep_model)
```

```
loss : 8.398951530456543
tp : 1647.0
fp : 12896.0
tn : 22609.0
fn : 1030.0
accuracy : 0.6352731585502625
precision : 0.11325035989284515
recall : 0.6152409315109253
auc : 0.6954426765441895
```

```
f1 Score 0.19128919860627178
```



Irrelevant Documents Detected (True Negatives): 22609  
Irrelevant Documents Incorrectly Detected (False Positives): 12896  
Relevant Documents Missed (False Negatives): 1030  
Relevant Documents Detected (True Positives): 1647  
Total Relevant Documents: 2677



```
[ ]: pred_labels = dense_deep_model.predict(test_features)
pred_labels = pred_labels.round(0)
pred_labels = pred_labels.astype(int)

dense_deep = {'Model': 'Dense Deep Model',
              'Loss': dense_deep_results[0],
              'Accuracy': accuracy_score(test_labels, pred_labels),
              'Precision': precision_score(test_labels, pred_labels),
              'Recall': recall_score(test_labels, pred_labels),
              'F1 Score': f1_score(test_labels, pred_labels)}

dense_deep
```

```
[ ]: {'Accuracy': 0.6352731653658792,
      'F1 Score': 0.19128919860627178,
      'Loss': 8.398951530456543,
      'Model': 'Dense Deep Model',
      'Precision': 0.11325036099841848,
      'Recall': 0.61524094135226}
```

### Predictions

```
[ ]: test_predictions = dense_deep_model.predict(test)
```

```
[ ]: test_predictions = test_predictions.round(0)
test_predictions = test_predictions.astype(int)
test_predictions
```

```
[ ]: array([[0],
           [0],
           [0],
           ...,
           [0],
           [0],
           [0]])
```

```
[ ]: predictions= pd.DataFrame(test_predictions)
predictions['Id'] = predictions.index
predictions.rename(columns={ predictions.columns[0]: "psrel" }, inplace = True)
predictions = predictions[['Id', 'psrel']]
predictions
```

```
[ ]:      Id  psrel
0      0      0
1      1      0
2      2      0
3      3      0
4      4      0
...    ...    ...
4995  4995      0
4996  4996      0
4997  4997      0
4998  4998      0
4999  4999      0
```

[5000 rows x 2 columns]

```
[ ]: predictions['psrel'].value_counts()
```

```
[ ]: 0    4846
     1     154
```

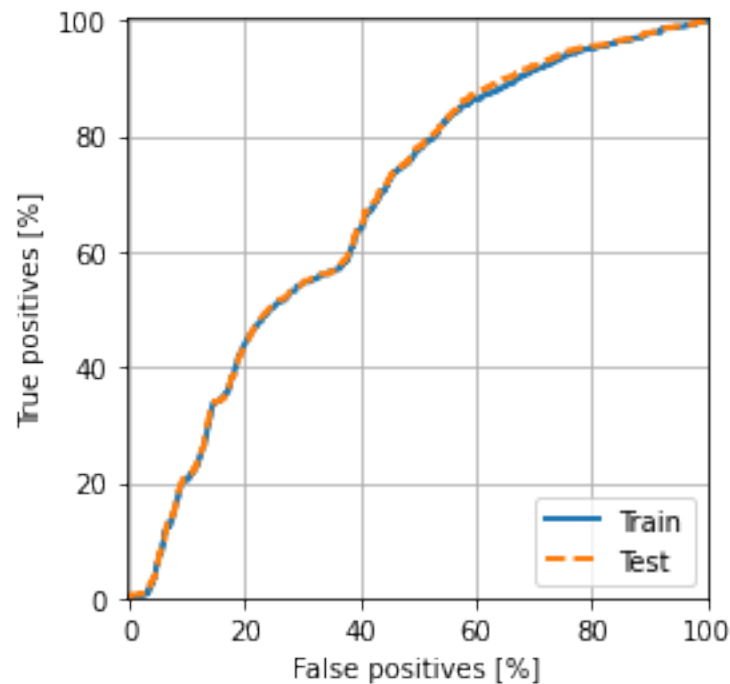
Name: psrel, dtype: int64

```
[ ]: predictions.to_csv('dense_deep_colab_predictions.csv', index=False)

[ ]: plot_roc("Train", train_labels, train_predictions_weighted, color=colors[0])
plot_roc("Test", test_labels, test_predictions_weighted, color=colors[1],
        linestyle='--')

plt.legend(loc='lower right')

[ ]: <matplotlib.legend.Legend at 0x7f75064c9d10>
```



### With Resampled Dataset

```
[ ]: resampled_dense_deep_model = build_dense_deep()
resampled_dense_deep_model.load_weights(initial_weights)

# Reset the bias to zero, since this dataset is balanced.
output_layer = resampled_dense_deep_model.layers[-1]
output_layer.bias.assign([0])

val_ds = tf.data.Dataset.from_tensor_slices((val_features, val_labels)).cache()
val_ds = val_ds.batch(BATCH_SIZE).prefetch(2)
```

```
[ ]: resampled_history = resampled_dense_deep_model.fit(  
    resampled_ds,  
    epochs=EPOCHS,  
    steps_per_epoch=resampled_steps_per_epoch,  
    callbacks=[early_stopping],  
    validation_data=val_ds)
```

Epoch 1/100

31/31 [=====] - 7s 78ms/step - loss: 14.5880 - tp: 10774.3438 - fp: 21328.6562 - tn: 31063.3438 - fn: 8743.6562 - accuracy: 0.5865 - precision: 0.3148 - recall: 0.5557 - auc: 0.6043 - val\_loss: 14.1993 - val\_tp: 1688.0000 - val\_fp: 19565.0000 - val\_tn: 8796.0000 - val\_fn: 496.0000 - val\_accuracy: 0.3432 - val\_precision: 0.0794 - val\_recall: 0.7729 - val\_auc: 0.6140

Epoch 2/100

31/31 [=====] - 1s 46ms/step - loss: 13.9093 - tp: 9287.8750 - fp: 8257.7188 - tn: 8541.5312 - fn: 7640.8750 - accuracy: 0.5278 - precision: 0.5282 - recall: 0.5479 - auc: 0.5381 - val\_loss: 13.4347 - val\_tp: 1654.0000 - val\_fp: 18796.0000 - val\_tn: 9565.0000 - val\_fn: 530.0000 - val\_accuracy: 0.3673 - val\_precision: 0.0809 - val\_recall: 0.7573 - val\_auc: 0.6336

Epoch 3/100

31/31 [=====] - 1s 43ms/step - loss: 13.2239 - tp: 9313.3438 - fp: 8249.5312 - tn: 8645.2188 - fn: 7519.9062 - accuracy: 0.5314 - precision: 0.5277 - recall: 0.5532 - auc: 0.5423 - val\_loss: 12.7215 - val\_tp: 1659.0000 - val\_fp: 17240.0000 - val\_tn: 11121.0000 - val\_fn: 525.0000 - val\_accuracy: 0.4184 - val\_precision: 0.0878 - val\_recall: 0.7596 - val\_auc: 0.6463

Epoch 4/100

31/31 [=====] - 1s 43ms/step - loss: 12.5606 - tp: 9411.9688 - fp: 8249.3125 - tn: 8686.1250 - fn: 7380.5938 - accuracy: 0.5365 - precision: 0.5315 - recall: 0.5617 - auc: 0.5500 - val\_loss: 12.0530 - val\_tp: 1616.0000 - val\_fp: 16119.0000 - val\_tn: 12242.0000 - val\_fn: 568.0000 - val\_accuracy: 0.4537 - val\_precision: 0.0911 - val\_recall: 0.7399 - val\_auc: 0.6497

Epoch 5/100

31/31 [=====] - 1s 44ms/step - loss: 11.9228 - tp: 9435.5938 - fp: 8052.0000 - tn: 8811.7500 - fn: 7428.6562 - accuracy: 0.5400 - precision: 0.5395 - recall: 0.5587 - auc: 0.5570 - val\_loss: 11.4267 - val\_tp: 1603.0000 - val\_fp: 14924.0000 - val\_tn: 13437.0000 - val\_fn: 581.0000 - val\_accuracy: 0.4924 - val\_precision: 0.0970 - val\_recall: 0.7340 - val\_auc: 0.6556

Epoch 6/100

31/31 [=====] - 1s 45ms/step - loss: 11.3310 - tp: 9502.2188 - fp: 8140.2188 - tn: 8733.1562 - fn: 7352.4062 - accuracy: 0.5388 - precision: 0.5367 - recall: 0.5617 - auc: 0.5520 - val\_loss: 10.8358 - val\_tp: 1578.0000 - val\_fp: 14371.0000 - val\_tn: 13990.0000 - val\_fn: 606.0000 - val\_accuracy: 0.5097 - val\_precision: 0.0989 - val\_recall: 0.7225 - val\_auc:

0.6586

Epoch 7/100

31/31 [=====] - 1s 44ms/step - loss: 10.7534 - tp: 9678.0625 - fp: 8104.1875 - tn: 8700.1875 - fn: 7245.5625 - accuracy: 0.5434 - precision: 0.5420 - recall: 0.5718 - auc: 0.5580 - val\_loss: 10.2789 - val\_tp: 1566.0000 - val\_fp: 13896.0000 - val\_tn: 14465.0000 - val\_fn: 618.0000 - val\_accuracy: 0.5248 - val\_precision: 0.1013 - val\_recall: 0.7170 - val\_auc: 0.6635

Epoch 8/100

31/31 [=====] - 1s 44ms/step - loss: 10.2039 - tp: 9652.8438 - fp: 8150.0000 - tn: 8798.4688 - fn: 7126.6875 - accuracy: 0.5480 - precision: 0.5421 - recall: 0.5768 - auc: 0.5593 - val\_loss: 9.7444 - val\_tp: 1541.0000 - val\_fp: 13392.0000 - val\_tn: 14969.0000 - val\_fn: 643.0000 - val\_accuracy: 0.5405 - val\_precision: 0.1032 - val\_recall: 0.7056 - val\_auc: 0.6646

Epoch 9/100

31/31 [=====] - 1s 43ms/step - loss: 9.6737 - tp: 9677.7500 - fp: 8053.6875 - tn: 8821.9375 - fn: 7174.6250 - accuracy: 0.5486 - precision: 0.5458 - recall: 0.5740 - auc: 0.5663 - val\_loss: 9.2399 - val\_tp: 1541.0000 - val\_fp: 13149.0000 - val\_tn: 15212.0000 - val\_fn: 643.0000 - val\_accuracy: 0.5485 - val\_precision: 0.1049 - val\_recall: 0.7056 - val\_auc: 0.6682

Epoch 10/100

31/31 [=====] - 1s 44ms/step - loss: 9.1797 - tp: 9745.1250 - fp: 8036.3125 - tn: 8755.7188 - fn: 7190.8438 - accuracy: 0.5477 - precision: 0.5468 - recall: 0.5743 - auc: 0.5611 - val\_loss: 8.7586 - val\_tp: 1535.0000 - val\_fp: 12866.0000 - val\_tn: 15495.0000 - val\_fn: 649.0000 - val\_accuracy: 0.5575 - val\_precision: 0.1066 - val\_recall: 0.7028 - val\_auc: 0.6689

Epoch 11/100

31/31 [=====] - 1s 46ms/step - loss: 8.6948 - tp: 9794.0000 - fp: 8126.5312 - tn: 8796.5938 - fn: 7010.8750 - accuracy: 0.5530 - precision: 0.5468 - recall: 0.5862 - auc: 0.5701 - val\_loss: 8.2989 - val\_tp: 1542.0000 - val\_fp: 12969.0000 - val\_tn: 15392.0000 - val\_fn: 642.0000 - val\_accuracy: 0.5544 - val\_precision: 0.1063 - val\_recall: 0.7060 - val\_auc: 0.6679

Epoch 12/100

31/31 [=====] - 1s 43ms/step - loss: 8.2461 - tp: 9798.4375 - fp: 8098.3438 - tn: 8724.5625 - fn: 7106.6562 - accuracy: 0.5488 - precision: 0.5477 - recall: 0.5785 - auc: 0.5657 - val\_loss: 7.8618 - val\_tp: 1549.0000 - val\_fp: 12897.0000 - val\_tn: 15464.0000 - val\_fn: 635.0000 - val\_accuracy: 0.5570 - val\_precision: 0.1072 - val\_recall: 0.7092 - val\_auc: 0.6740

Epoch 13/100

31/31 [=====] - 1s 44ms/step - loss: 7.8074 - tp: 10076.7188 - fp: 8077.2188 - tn: 8694.2812 - fn: 6879.7812 - accuracy: 0.5567 - precision: 0.5550 - recall: 0.5941 - auc: 0.5733 - val\_loss: 7.4464 - val\_tp: 1542.0000 - val\_fp: 12920.0000 - val\_tn: 15441.0000 - val\_fn: 642.0000 -

val\_accuracy: 0.5560 - val\_precision: 0.1066 - val\_recall: 0.7060 - val\_auc: 0.6704

Epoch 14/100

31/31 [=====] - 1s 45ms/step - loss: 7.3888 - tp: 9936.7188 - fp: 8056.1875 - tn: 8950.4062 - fn: 6784.6875 - accuracy: 0.5601 - precision: 0.5511 - recall: 0.5951 - auc: 0.5803 - val\_loss: 7.0450 - val\_tp: 1519.0000 - val\_fp: 12618.0000 - val\_tn: 15743.0000 - val\_fn: 665.0000 - val\_accuracy: 0.5651 - val\_precision: 0.1074 - val\_recall: 0.6955 - val\_auc: 0.6717

Epoch 15/100

31/31 [=====] - 1s 45ms/step - loss: 6.9973 - tp: 9999.6250 - fp: 8007.9688 - tn: 8896.9062 - fn: 6823.5000 - accuracy: 0.5617 - precision: 0.5570 - recall: 0.5959 - auc: 0.5795 - val\_loss: 6.6669 - val\_tp: 1521.0000 - val\_fp: 12598.0000 - val\_tn: 15763.0000 - val\_fn: 663.0000 - val\_accuracy: 0.5659 - val\_precision: 0.1077 - val\_recall: 0.6964 - val\_auc: 0.6731

Epoch 16/100

31/31 [=====] - 1s 44ms/step - loss: 6.6257 - tp: 9946.8125 - fp: 8059.6562 - tn: 8861.3750 - fn: 6860.1562 - accuracy: 0.5574 - precision: 0.5529 - recall: 0.5893 - auc: 0.5779 - val\_loss: 6.3084 - val\_tp: 1520.0000 - val\_fp: 12694.0000 - val\_tn: 15667.0000 - val\_fn: 664.0000 - val\_accuracy: 0.5627 - val\_precision: 0.1069 - val\_recall: 0.6960 - val\_auc: 0.6760

Epoch 17/100

31/31 [=====] - 1s 43ms/step - loss: 6.2696 - tp: 10071.4375 - fp: 8054.4062 - tn: 8843.3750 - fn: 6758.7812 - accuracy: 0.5610 - precision: 0.5554 - recall: 0.5990 - auc: 0.5781 - val\_loss: 5.9717 - val\_tp: 1534.0000 - val\_fp: 12791.0000 - val\_tn: 15570.0000 - val\_fn: 650.0000 - val\_accuracy: 0.5600 - val\_precision: 0.1071 - val\_recall: 0.7024 - val\_auc: 0.6777

Epoch 18/100

31/31 [=====] - 1s 43ms/step - loss: 5.9266 - tp: 10224.3125 - fp: 7987.7812 - tn: 8896.7500 - fn: 6619.1562 - accuracy: 0.5676 - precision: 0.5630 - recall: 0.6068 - auc: 0.5856 - val\_loss: 5.6477 - val\_tp: 1549.0000 - val\_fp: 12811.0000 - val\_tn: 15550.0000 - val\_fn: 635.0000 - val\_accuracy: 0.5598 - val\_precision: 0.1079 - val\_recall: 0.7092 - val\_auc: 0.6799

Epoch 19/100

31/31 [=====] - 1s 44ms/step - loss: 5.6045 - tp: 10369.7812 - fp: 8019.5625 - tn: 8811.5625 - fn: 6527.0938 - accuracy: 0.5690 - precision: 0.5616 - recall: 0.6150 - auc: 0.5885 - val\_loss: 5.3459 - val\_tp: 1569.0000 - val\_fp: 12973.0000 - val\_tn: 15388.0000 - val\_fn: 615.0000 - val\_accuracy: 0.5551 - val\_precision: 0.1079 - val\_recall: 0.7184 - val\_auc: 0.6784

Epoch 20/100

31/31 [=====] - 1s 46ms/step - loss: 5.2986 - tp: 10395.1562 - fp: 8114.6562 - tn: 8773.9688 - fn: 6444.2188 - accuracy: 0.5681 - precision: 0.5610 - recall: 0.6163 - auc: 0.5868 - val\_loss: 5.0589 - val\_tp:

1603.0000 - val\_fp: 13188.0000 - val\_tn: 15173.0000 - val\_fn: 581.0000 -  
val\_accuracy: 0.5492 - val\_precision: 0.1084 - val\_recall: 0.7340 - val\_auc:  
0.6814

Epoch 21/100

31/31 [=====] - 1s 45ms/step - loss: 5.0080 - tp:  
10642.5625 - fp: 8125.6250 - tn: 8641.0938 - fn: 6318.7188 - accuracy: 0.5716 -  
precision: 0.5670 - recall: 0.6282 - auc: 0.5908 - val\_loss: 4.7854 - val\_tp:  
1629.0000 - val\_fp: 13504.0000 - val\_tn: 14857.0000 - val\_fn: 555.0000 -  
val\_accuracy: 0.5397 - val\_precision: 0.1076 - val\_recall: 0.7459 - val\_auc:  
0.6794

Epoch 22/100

31/31 [=====] - 1s 43ms/step - loss: 4.7329 - tp:  
10647.7812 - fp: 8137.6562 - tn: 8738.5000 - fn: 6204.0625 - accuracy: 0.5738 -  
precision: 0.5650 - recall: 0.6315 - auc: 0.5931 - val\_loss: 4.5285 - val\_tp:  
1665.0000 - val\_fp: 13923.0000 - val\_tn: 14438.0000 - val\_fn: 519.0000 -  
val\_accuracy: 0.5272 - val\_precision: 0.1068 - val\_recall: 0.7624 - val\_auc:  
0.6856

Epoch 23/100

31/31 [=====] - 1s 43ms/step - loss: 4.4725 - tp:  
10772.8438 - fp: 8138.7188 - tn: 8721.4688 - fn: 6094.9688 - accuracy: 0.5789 -  
precision: 0.5720 - recall: 0.6392 - auc: 0.5959 - val\_loss: 4.2840 - val\_tp:  
1699.0000 - val\_fp: 14227.0000 - val\_tn: 14134.0000 - val\_fn: 485.0000 -  
val\_accuracy: 0.5184 - val\_precision: 0.1067 - val\_recall: 0.7779 - val\_auc:  
0.6855

Epoch 24/100

31/31 [=====] - 1s 44ms/step - loss: 4.2255 - tp:  
10762.0938 - fp: 8205.4688 - tn: 8697.7812 - fn: 6062.6562 - accuracy: 0.5760 -  
precision: 0.5658 - recall: 0.6380 - auc: 0.5964 - val\_loss: 4.0532 - val\_tp:  
1763.0000 - val\_fp: 14770.0000 - val\_tn: 13591.0000 - val\_fn: 421.0000 -  
val\_accuracy: 0.5027 - val\_precision: 0.1066 - val\_recall: 0.8072 - val\_auc:  
0.6820

Epoch 25/100

31/31 [=====] - 1s 44ms/step - loss: 3.9879 - tp:  
10918.8125 - fp: 8165.8438 - tn: 8688.0625 - fn: 5955.2812 - accuracy: 0.5820 -  
precision: 0.5728 - recall: 0.6472 - auc: 0.6047 - val\_loss: 3.8358 - val\_tp:  
1808.0000 - val\_fp: 15262.0000 - val\_tn: 13099.0000 - val\_fn: 376.0000 -  
val\_accuracy: 0.4880 - val\_precision: 0.1059 - val\_recall: 0.8278 - val\_auc:  
0.6861

Epoch 26/100

31/31 [=====] - 1s 45ms/step - loss: 3.7656 - tp:  
10972.8750 - fp: 8131.6562 - tn: 8759.3750 - fn: 5864.0938 - accuracy: 0.5841 -  
precision: 0.5738 - recall: 0.6494 - auc: 0.6092 - val\_loss: 3.6288 - val\_tp:  
1823.0000 - val\_fp: 15494.0000 - val\_tn: 12867.0000 - val\_fn: 361.0000 -  
val\_accuracy: 0.4809 - val\_precision: 0.1053 - val\_recall: 0.8347 - val\_auc:  
0.6787

Epoch 27/100

31/31 [=====] - 1s 44ms/step - loss: 3.5579 - tp:  
11165.3125 - fp: 8160.1562 - tn: 8682.9375 - fn: 5719.5938 - accuracy: 0.5876 -

```

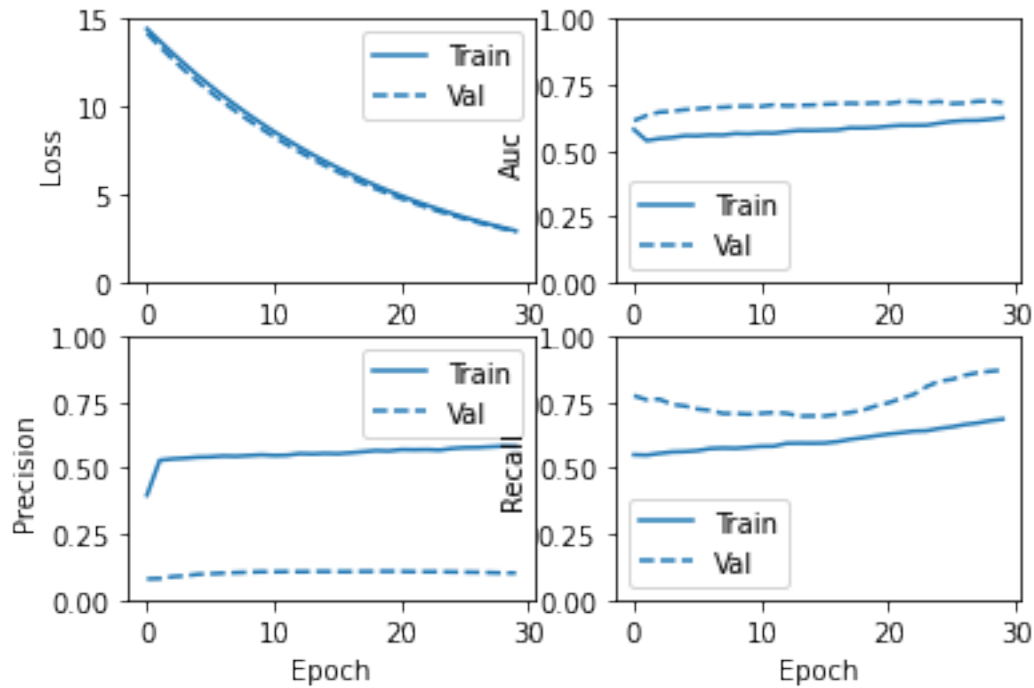
precision: 0.5776 - recall: 0.6601 - auc: 0.6114 - val_loss: 3.4350 - val_tp:
1856.0000 - val_fp: 15883.0000 - val_tn: 12478.0000 - val_fn: 328.0000 -
val_accuracy: 0.4693 - val_precision: 0.1046 - val_recall: 0.8498 - val_auc:
0.6807
Epoch 28/100
31/31 [=====] - 1s 43ms/step - loss: 3.3642 - tp:
11304.6875 - fp: 8239.3438 - tn: 8598.7188 - fn: 5585.2500 - accuracy: 0.5884 -
precision: 0.5775 - recall: 0.6682 - auc: 0.6107 - val_loss: 3.2533 - val_tp:
1878.0000 - val_fp: 16313.0000 - val_tn: 12048.0000 - val_fn: 306.0000 -
val_accuracy: 0.4559 - val_precision: 0.1032 - val_recall: 0.8599 - val_auc:
0.6870
Epoch 29/100
31/31 [=====] - 1s 43ms/step - loss: 3.1755 - tp:
11461.0625 - fp: 8191.7188 - tn: 8603.3750 - fn: 5471.8438 - accuracy: 0.5942 -
precision: 0.5838 - recall: 0.6758 - auc: 0.6177 - val_loss: 3.0824 - val_tp:
1890.0000 - val_fp: 16669.0000 - val_tn: 11692.0000 - val_fn: 294.0000 -
val_accuracy: 0.4447 - val_precision: 0.1018 - val_recall: 0.8654 - val_auc:
0.6879
Epoch 30/100
31/31 [=====] - 1s 43ms/step - loss: 2.9993 - tp:
11445.4688 - fp: 8232.4375 - tn: 8750.4375 - fn: 5299.6562 - accuracy: 0.5979 -
precision: 0.5813 - recall: 0.6831 - auc: 0.6226 - val_loss: 2.9204 - val_tp:
1900.0000 - val_fp: 16773.0000 - val_tn: 11588.0000 - val_fn: 284.0000 -
val_accuracy: 0.4416 - val_precision: 0.1018 - val_recall: 0.8700 - val_auc:
0.6823
Restoring model weights from the end of the best epoch.
Epoch 00030: early stopping

```

## Evaluation

```
[ ]: plot_metrics(resampled_history)
```





```
[ ]: !mkdir -p saved_model
resampled_dense_deep_model.save('saved_model/resampled_dense_deep_model')
```

INFO:tensorflow:Assets written to: saved\_model/resampled\_dense\_deep\_model/assets

```
[ ]: dd_train_predictions_resampled = resampled_dense_deep_model.
      ↳predict(train_features, batch_size=BATCH_SIZE)
dd_test_predictions_resampled = resampled_dense_deep_model.
      ↳predict(test_features, batch_size=BATCH_SIZE)
```

```
[ ]: resampled_results = resampled_dense_deep_model.evaluate(test_features,
      ↳test_labels,
                                                    batch_size=BATCH_SIZE, verbose=0)
for name, value in zip(resampled_dense_deep_model.metrics_names,
      ↳resampled_results):
    print(name, ': ', value)
print()

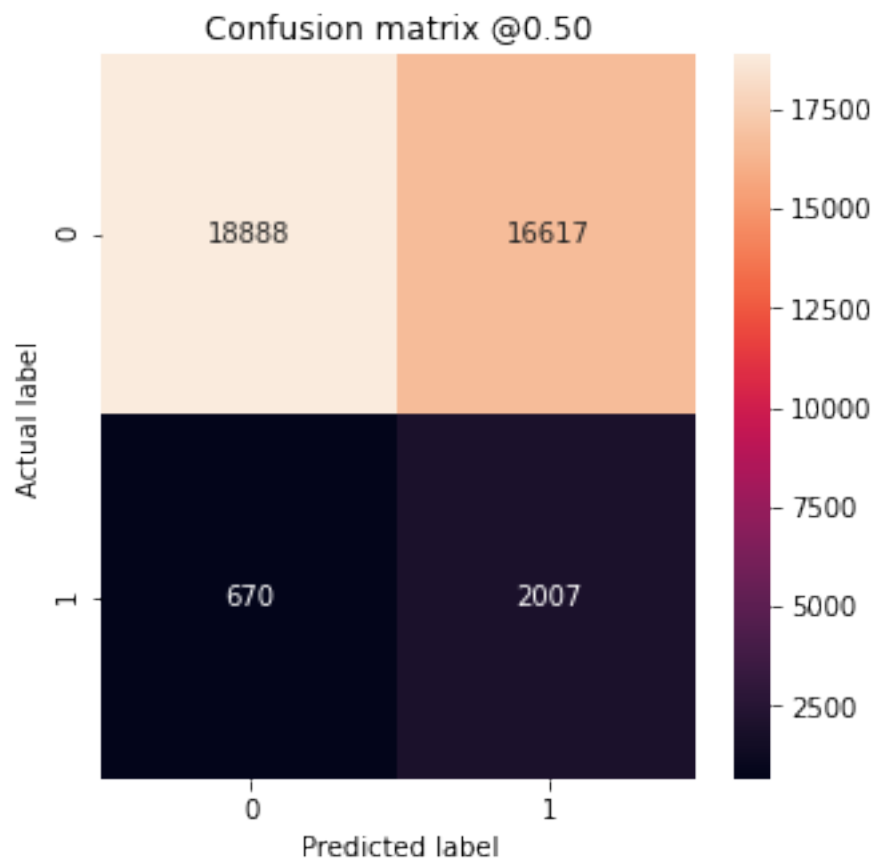
plot_cm(test_labels, dd_test_predictions_resampled)
```

```
loss : 5.059300422668457
tp : 2007.0
fp : 16617.0
tn : 18888.0
fn : 670.0
```

accuracy : 0.5472474098205566  
precision : 0.10776417702436447  
recall : 0.7497198581695557  
auc : 0.6902697086334229

f1 Score 0.1884418571898033

Irrelevant Documents Detected (True Negatives): 18888  
Irrelevant Documents Incorrectly Detected (False Positives): 16617  
Relevant Documents Missed (False Negatives): 670  
Relevant Documents Detected (True Positives): 2007  
Total Relevant Documents: 2677



```
[ ]: pred_labels = resampled_dense_deep_model.predict(test_features)
     pred_labels = pred_labels.round(0)
     pred_labels = pred_labels.astype(int)

     deep_dense_resampled = {'Model': 'Resampled Dense Deep Layer Model',
                             'Loss': resampled_results[0],
```

```

    'Accuracy': accuracy_score(test_labels, pred_labels),
    'Precision': precision_score(test_labels, pred_labels),
    'Recall': recall_score(test_labels, pred_labels),
    'F1 Score': f1_score(test_labels, pred_labels)}

```

```
deep_dense_resampled
```

```

[ ]: {'Accuracy': 0.5472473940600283,
      'F1 Score': 0.1884418571898033,
      'Loss': 5.059300422668457,
      'Model': 'Resampled Dense Deep Layer Model',
      'Precision': 0.10776417525773196,
      'Recall': 0.749719835636907}

```

### Predictions

```
[ ]: test_predictions = resampled_dense_deep_model.predict(test)
```

```

[ ]: test_predictions = test_predictions.round(0)
test_predictions = test_predictions.astype(int)
test_predictions

```

```

[ ]: array([[0],
           [0],
           [0],
           ...,
           [0],
           [0],
           [0]])

```

```

[ ]: predictions= pd.DataFrame(test_predictions)
predictions['Id'] = predictions.index
predictions.rename(columns={ predictions.columns[0]: "psrel" }, inplace = True)
predictions = predictions[['Id', 'psrel']]
predictions

```

```

[ ]:
      Id  psrel
0      0      0
1      1      0
2      2      0
3      3      0
4      4      0
...    ...    ...
4995  4995      0
4996  4996      0
4997  4997      0
4998  4998      0
4999  4999      0

```

[5000 rows x 2 columns]

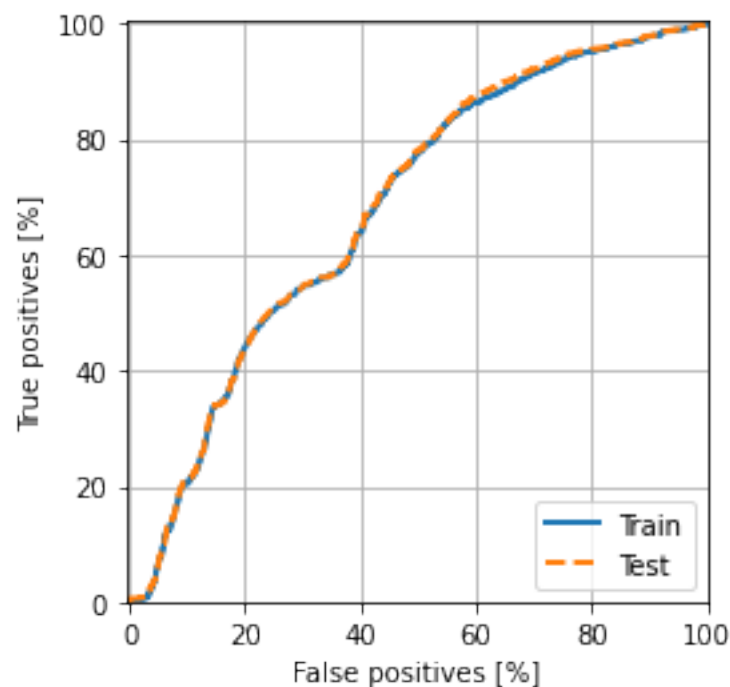
```
[ ]: predictions['psrel'].value_counts()
```

```
[ ]: 0    4215  
     1     785  
     Name: psrel, dtype: int64
```

```
[ ]: predictions.to_csv('resampled dense deep colab predictions.csv', index=False)
```

```
[ ]: plot_roc("Train", train_labels, train_predictions_weighted, color=colors[0])  
     plot_roc("Test", test_labels, test_predictions_weighted, color=colors[1],  
             ↳linestyle='--')  
  
     plt.legend(loc='lower right')
```

```
[ ]: <matplotlib.legend.Legend at 0x7f7503e7bd90>
```



## 6.2.4 Wide and Deep Model

```
[ ]: input_wide = keras.layers.Input(shape=[23], name="wide_input")  
     input_deep = keras.layers.Input(shape=[74], name="deep_input")
```

```
[ ]: wd_early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    verbose=1,
    patience=10,
    mode='min',
    restore_best_weights=True)
```

## Wide and Deep Model Class

```
[ ]: # Wide and Deep Learning Model function
class WideAndDeepModel(keras.Model):
    def __init__(self, n_neurons=30, activation="elu", **kwargs):
        super().__init__(**kwargs)
        self.hidden1 = WDRegularizedDense(n_neurons)
        self.hidden2 = WDRegularizedDense(n_neurons)
        self.main_output = keras.layers.Dense(1, activation="sigmoid")

    def call(self, inputs):
        input_wide, input_deep = inputs
        hidden1 = self.hidden1(input_deep)
        hidden2 = self.hidden2(hidden1)
        concat = keras.layers.concatenate([input_wide, hidden2])
        main_output = self.main_output(concat)

        return main_output
```

```
[ ]: wide_train_features.shape
```

```
[ ]: (122179, 23)
```

```
[ ]: deep_train_features.shape
```

```
[ ]: (122179, 74)
```

```
[ ]: wd_model = WideAndDeepModel(n_neurons=30, activation="selu")

wd_model.compile(loss=["binary_crossentropy"], optimizer=keras.optimizers.
    ↳Nadam(lr=3e-4), metrics="accuracy")

wd_model_history = wd_model.fit([wide_train_features, deep_train_features],
    ↳train_labels,
                                epochs=40,
                                validation_data=([wide_val_features, deep_val_features],
    ↳val_labels),
                                callbacks=[wd_early_stopping],
                                verbose=1)

total_loss = wd_model.evaluate((wide_test_features, deep_test_features),
```

```

test_labels)
print(total_loss)

Epoch 1/40
3819/3819 [=====] - 8s 2ms/step - loss: 2.6609 -
accuracy: 0.8517 - val_loss: 0.2607 - val_accuracy: 0.9285
Epoch 2/40
3819/3819 [=====] - 7s 2ms/step - loss: 0.2645 -
accuracy: 0.9266 - val_loss: 0.2594 - val_accuracy: 0.9285
Epoch 3/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2616 -
accuracy: 0.9274 - val_loss: 0.2588 - val_accuracy: 0.9285
Epoch 4/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2603 -
accuracy: 0.9278 - val_loss: 0.2586 - val_accuracy: 0.9285
Epoch 5/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2578 -
accuracy: 0.9288 - val_loss: 0.2587 - val_accuracy: 0.9285
Epoch 6/40
3819/3819 [=====] - 7s 2ms/step - loss: 0.2643 -
accuracy: 0.9262 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 7/40
3819/3819 [=====] - 7s 2ms/step - loss: 0.2594 -
accuracy: 0.9281 - val_loss: 0.2585 - val_accuracy: 0.9285
Epoch 8/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2621 -
accuracy: 0.9270 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 9/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2618 -
accuracy: 0.9272 - val_loss: 0.2585 - val_accuracy: 0.9285
Epoch 10/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2652 -
accuracy: 0.9257 - val_loss: 0.2587 - val_accuracy: 0.9285
Epoch 11/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2589 -
accuracy: 0.9282 - val_loss: 0.2585 - val_accuracy: 0.9285
Epoch 12/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2614 -
accuracy: 0.9273 - val_loss: 0.2585 - val_accuracy: 0.9285
Epoch 13/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2644 -
accuracy: 0.9261 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 14/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2589 -
accuracy: 0.9283 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 15/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2589 -
accuracy: 0.9282 - val_loss: 0.2587 - val_accuracy: 0.9285

```

```

Epoch 16/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2608 -
accuracy: 0.9276 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 17/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2604 -
accuracy: 0.9277 - val_loss: 0.2585 - val_accuracy: 0.9285
Epoch 18/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2623 -
accuracy: 0.9270 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 19/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2592 -
accuracy: 0.9281 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 20/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2645 -
accuracy: 0.9261 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 21/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2573 -
accuracy: 0.9289 - val_loss: 0.2603 - val_accuracy: 0.9285
Epoch 22/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2590 -
accuracy: 0.9283 - val_loss: 0.2584 - val_accuracy: 0.9285
Epoch 23/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2592 -
accuracy: 0.9282 - val_loss: 0.2585 - val_accuracy: 0.9285
Epoch 24/40
3819/3819 [=====] - 7s 2ms/step - loss: 0.2583 -
accuracy: 0.9285 - val_loss: 0.2591 - val_accuracy: 0.9285
Epoch 25/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2592 -
accuracy: 0.9282 - val_loss: 0.2589 - val_accuracy: 0.9285
Epoch 26/40
3819/3819 [=====] - 6s 2ms/step - loss: 0.2599 -
accuracy: 0.9279 - val_loss: 0.2587 - val_accuracy: 0.9285
Restoring model weights from the end of the best epoch.
Epoch 00026: early stopping
1194/1194 [=====] - 1s 1ms/step - loss: 0.2549 -
accuracy: 0.9299
[0.2549344301223755, 0.9298884272575378]

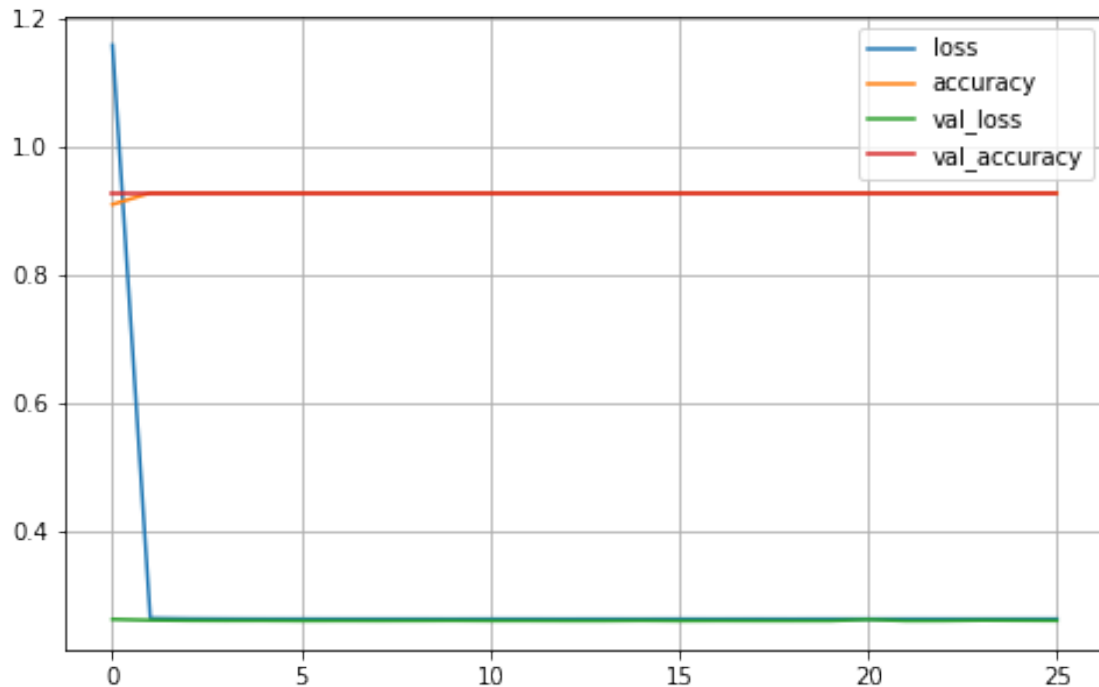
```

## Evaluation

```

[ ]: pd.DataFrame(wd_model_history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.show()

```



```
[ ]: train_predictions_wd_model = wd_model.predict((wide_train_features,
    ↳deep_train_features))
test_predictions_wd_model = wd_model.predict((wide_test_features,
    ↳deep_test_features))
```

```
[ ]: wd_results = wd_model.evaluate((wide_test_features, deep_test_features),
    ↳test_labels, verbose=0)
for name, value in zip(wd_model.metrics_names, wd_results):
    print(name, ': ', value)
print()

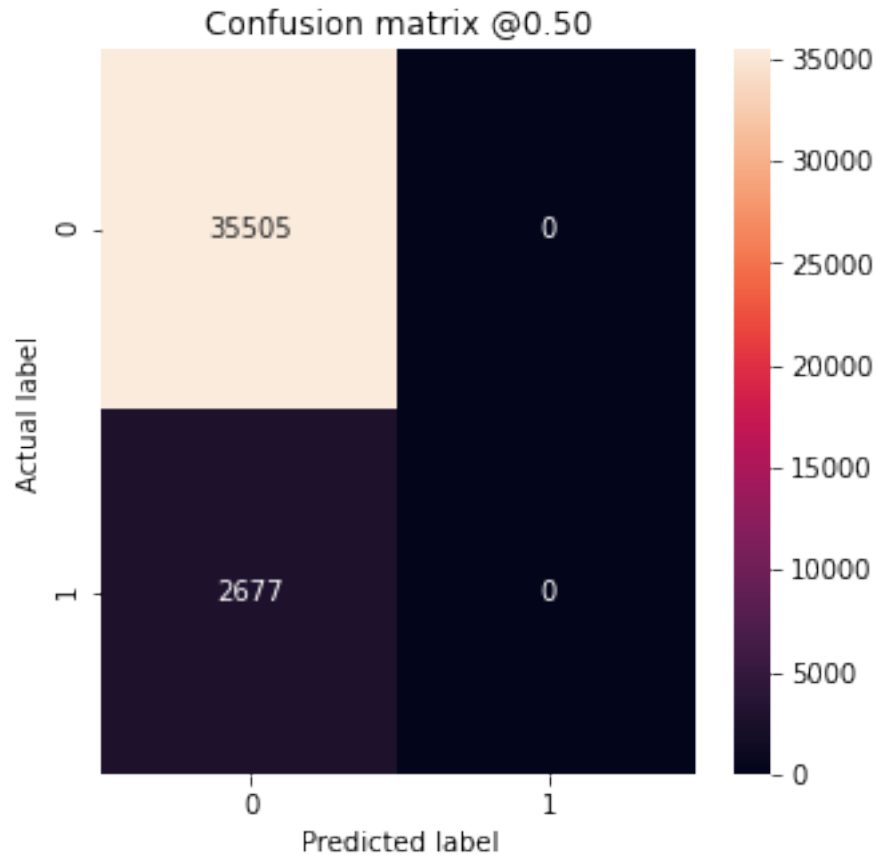
plot_cm(test_labels, test_predictions_wd_model)
```

```
loss : 0.2549344301223755
accuracy : 0.9298884272575378
```

```
f1 Score 0.0
```

```
Irrelevant Documents Detected (True Negatives): 35505
Irrelevant Documents Incorrectly Detected (False Positives): 0
Relevant Documents Missed (False Negatives): 2677
Relevant Documents Detected (True Positives): 0
Total Relevant Documents: 2677
```





```
[ ]: pred_labels = wd_model.predict((wide_test_features, deep_test_features))
pred_labels = pred_labels.round(0)
pred_labels = pred_labels.astype(int)
```

```
wd = {'Model': 'Wide and Deep Model',
      'Loss': wd_results[0],
      'Accuracy': accuracy_score(test_labels, pred_labels),
      'Precision': precision_score(test_labels, pred_labels),
      'Recall': recall_score(test_labels, pred_labels),
      'F1 Score': f1_score(test_labels, pred_labels)}
```

```
wd
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/metrics/_classification.py:1272:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 due to no
predicted samples. Use `zero_division` parameter to control this behavior.
_warn_prf(average, modifier, msg_start, len(result))
```

```
[ ]: {'Accuracy': 0.9298884291027185,
      'F1 Score': 0.0,
      'Loss': 0.2549344301223755,
      'Model': 'Wide and Deep Model',
      'Precision': 0.0,
      'Recall': 0.0}
```

### Predictions

```
[ ]: test_predictions = wd_model.predict((wide_test, deep_test))
test_predictions
```

```
[ ]: array([[0.06221834],
            [0.05450538],
            [0.05234206],
            ...,
            [0.07361767],
            [0.06896427],
            [0.06474838]], dtype=float32)
```

```
[ ]: test_predictions = test_predictions.round(0)
test_predictions = test_predictions.astype(int)
test_predictions
```

```
[ ]: array([[0],
            [0],
            [0],
            ...,
            [0],
            [0],
            [0]])
```

```
[ ]: predictions= pd.DataFrame(test_predictions)
predictions['Id'] = predictions.index
predictions.rename(columns={ predictions.columns[0]: "psrel" }, inplace = True)
predictions = predictions[['Id', 'psrel']]
predictions
```

```
[ ]:      Id  psrel
0      0      0
1      1      0
2      2      0
3      3      0
4      4      0
...    ...    ...
4995  4995      0
4996  4996      0
4997  4997      0
```

```
4998  4998      0
4999  4999      0
```

```
[5000 rows x 2 columns]
```

```
[ ]: predictions['psrel'].value_counts()
```

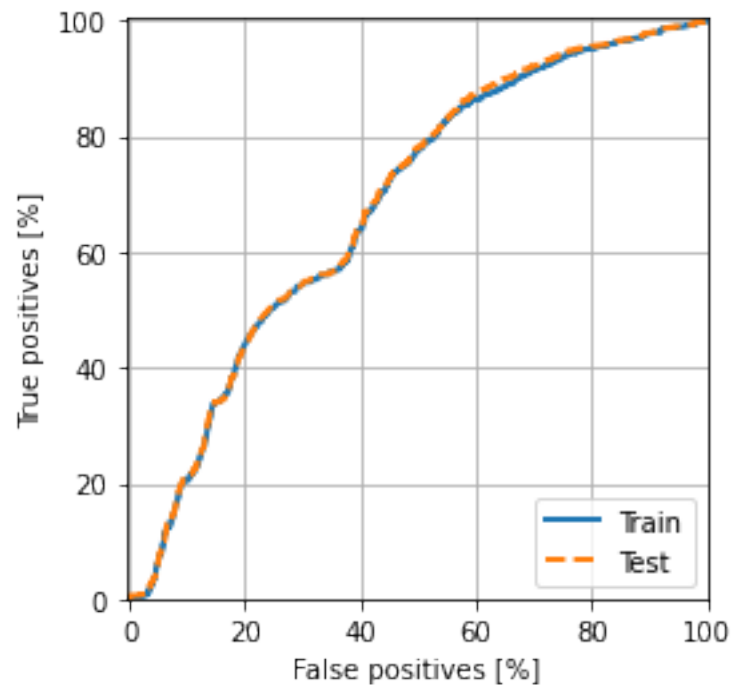
```
[ ]: 0    5000
      Name: psrel, dtype: int64
```

```
[ ]: predictions.to_csv('wide and deep colab predictions.csv', index=False)
```

```
[ ]: plot_roc("Train", train_labels, train_predictions_weighted, color=colors[0])
      plot_roc("Test", test_labels, test_predictions_weighted, color=colors[1],
      →linestyle='--')

      plt.legend(loc='lower right')
```

```
[ ]: <matplotlib.legend.Legend at 0x7f75011567d0>
```



### 6.3 Any Additional Analysis

- Add in any additional analysis etc that you performed here.

```
[ ]: ### Create table of testing performance
```

```
model_dict = [unwt_rnd, wt_rnd, tl_bias, tl_wt, tl_resampled, dense_deep,   
              ↪ deep_dense_resampled, wd]
```

```
[ ]: model_df = pd.DataFrame(model_dict)  
model_df
```

```
[ ]: 

|   | Model                              | Loss     | ... | Recall   | F1 Score |
|---|------------------------------------|----------|-----|----------|----------|
| 0 | Unweighted Random Forest           | 0.000000 | ... | 0.005230 | 0.010405 |
| 1 | Weighted Random Forest             | 0.000000 | ... | 0.898020 | 0.283993 |
| 2 | Adjusted Bias Three Layer Model    | 8.092644 | ... | 0.000000 | 0.000000 |
| 3 | Adjusted Weights Three Layer Model | 5.860197 | ... | 0.200598 | 0.167342 |
| 4 | Resampled Three Layer Model        | 1.794922 | ... | 0.861412 | 0.186140 |
| 5 | Dense Deep Model                   | 8.398952 | ... | 0.615241 | 0.191289 |
| 6 | Resampled Dense Deep Layer Model   | 5.059300 | ... | 0.749720 | 0.188442 |
| 7 | Wide and Deep Model                | 0.254934 | ... | 0.000000 | 0.000000 |


```

```
[8 rows x 6 columns]
```

```
[ ]: plot_roc("TL Train Baseline", train_labels, train_predictions_baseline,   
             ↪ color=colors[0])  
plot_roc("TL Test Baseline", test_labels, test_predictions_baseline,   
         ↪ color=colors[0], linestyle='--')  
  
plot_roc("TL Train Weighted", train_labels, train_predictions_weighted,   
         ↪ color=colors[1])  
plot_roc("TL Test Weighted", test_labels, test_predictions_weighted,   
         ↪ color=colors[1], linestyle='--')  
  
plot_roc("TL Train Resampled", train_labels, train_predictions_resampled,   
         ↪ color=colors[2])  
plot_roc("TL Test Resampled", test_labels, test_predictions_resampled,   
         ↪ color=colors[2], linestyle='--')  
  
plot_roc("DD Train Weighted", train_labels, train_predictions_dense_deep_model,   
         ↪ color=colors[3])  
plot_roc("DD Test Weighted", test_labels, test_predictions_dense_deep_model,   
         ↪ color=colors[3], linestyle='--')  
  
plot_roc("DD Train Resampled", train_labels, dd_train_predictions_resampled,   
         ↪ color=colors[4])  
plot_roc("DD Test Resampled", test_labels, dd_test_predictions_resampled,   
         ↪ color=colors[4], linestyle='--')  
  
plot_roc("W&D Train", train_labels, train_predictions_wd_model, color=colors[5])
```

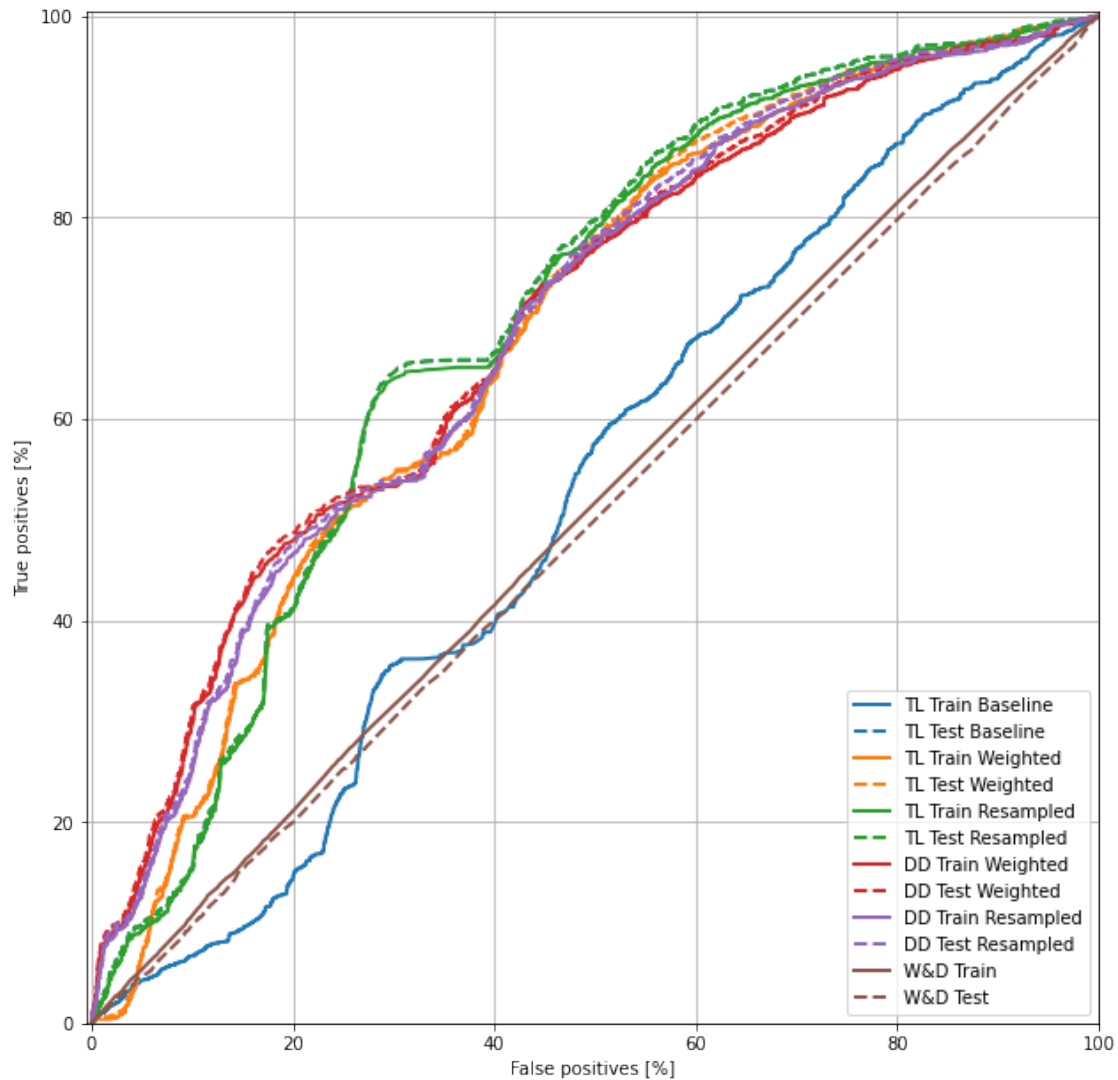
```

plot_roc("W&D Test", test_labels, test_predictions_wd_model, color=colors[5],
        linestyle='--')

plt.legend(loc='lower right')

fig = plt.gcf()
fig.set_size_inches(18.5, 10.5)
fig.savefig('graph.png', dpi=100)

```



```

[ ]: TL_HP = {"Model": "Three Layer Model",
              "Neurons per Layer": "30",
              "Number of Layers": "3",

```

```

"Kernel Initializer": "He Initialization",
"Activation Function": "ELU",
"Normalization": "Batch Normalisation",
"Regularisation": "l1 and l2 regularisation with Dropout if needed",
"Optimizer": "Nadam",
"Learning Rate Schedule": "1cycle"}

```

```

DD_HP = {"Model": "Dense Deep Model",
"Neurons per Layer": "30",
"Number of Layers": "5",
"Kernel Initializer": "He Initialization",
"Activation Function": "ELU",
"Normalization": "Batch Normalisation",
"Regularisation": "l1 and l2 regularisation with Dropout if needed",
"Optimizer": "Nadam",
"Learning Rate Schedule": "1cycle"}

```

```

WD_HP = {"Model": "Wide and Deep Model",
"Neurons per Layer": "30",
"Number of Layers": "5 (2 inputs, 2 dense and output)",
"Kernel Initializer": "LeCun Initialization",
"Activation Function": "SELU",
"Normalization": "None (Self-Normalisation)",
"Regularisation": "l1 and l2 regularisation",
"Optimizer": "Nadam",
"Learning Rate Schedule": "1cycle"}

```

```
[ ]: hyp_dict = [TL_HP, DD_HP, WD_HP]
```

```

hyp_df = pd.DataFrame(hyp_dict)
hyp_df

```

```
[ ]:
```

	Model	Neurons per Layer	...	Optimizer	Learning Rate Schedule
0	Three Layer Model	30	...	Nadam	1cycle
1	Dense Deep Model	30	...	Nadam	1cycle
2	Wide and Deep Model	30	...	Nadam	1cycle

```
[3 rows x 9 columns]
```