```
import tensorflow as tf
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import csv
from sklearn.model_selection import train_test_split, StratifiedKFold, GridSearchCV
from tensorflow import keras
from keras.models import Sequential
import keras_tuner as kt
from keras.layers import Dense
from keras import layers
import keras_tuner
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
from sklearn import svm
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.metrics import f1_score
from sklearn.metrics import recall_score, accuracy_score
from sklearn.utils import resample
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score
```

```
#Read in the csv as strokeData
strokeData = pd.read_csv('healthcare-dataset-stroke-data.csv')
```

```
#Analyze the shape ofthe data
print("Number of Samples: ", strokeData.shape[0])
print("Number of Features: ", strokeData.shape[1])

#Also, inspect the beginning of the data frame
strokeData.head(20)
```

```
Number of Samples:  5110
Number of Features:  12
```

	id	gender	age	hypertension	heart_disease	ever_mar
0	9046	Male	67.0	0	1	
1	51676	Female	61.0	0	0	
2	31112	Male	80.0	0	1	
3	60182	Female	49.0	0	0	
4	1665	Female	79.0	1	0	
5	56669	Male	81.0	0	0	
6	53882	Male	74.0	1	1	

```
#Remove the id column as it will not be used as a predictor for stroke
strokeData.drop('id',axis=1, inplace=True)
```

```
#Check the df again
strokeData
```

	gender	age	hypertension	heart_disease	ever_married
0	Male	67.0	0	1	Yes
1	Female	61.0	0	0	Yes
2	Male	80.0	0	1	Yes
3	Female	49.0	0	0	Yes
4	Female	79.0	1	0	Yes
...	...	...	...	...	...
5105	Female	80.0	1	0	Yes

Lets also note what the integer codes correspond to.

Gender:

- 0 = male
- 1 = female

Ever_Married:

- 0 = Yes
- 1 = No

Work_Type:

- 0 = Private
- 1 = Self-Employed
- 2 = Govt Job
- 3 = Children (too young)
- 4 = Never Worked

Residence_Type:

- 0 = Urban
- 1 = Rural

Smoking_Status:

- 0 = formerly smoked
- 1 = never smoked
- 2 = smokes
- 3 = unknown

```
#Now, lets remove all NAs that may make calculations difficult
#It appears that some patients did not report their BMI, and thus cannot be used in our neural network; shown by NA

#Before we run dropna(), lets convert empty? (0) cells to NaN so dropna() can work properly
strokeData = strokeData.replace('', np.nan)

strokeData.dropna(inplace=True) #inplace is true because it defaults to false and would have to be assigned to another
strokeData.head(20)
```

	gender	age	hypertension	heart_disease	ever_married	w
0	Male	67.0	0	1	Yes	
2	Male	80.0	0	1	Yes	
3	Female	49.0	0	0	Yes	
4	Female	79.0	1	0	Yes	
5	Male	81.0	0	0	Yes	

```
#Also, lets remove columns of smokers that we do not know the history of.
#Any entries in smoking_status of "unknown" will be removed (integer code of 3)

strokeData = strokeData[strokeData.smoking_status != "Unknown"]
strokeData.head(20)
```

	gender	age	hypertension	heart_disease	ever_married	w
0	Male	67.0	0	1	Yes	
2	Male	80.0	0	1	Yes	
3	Female	49.0	0	0	Yes	
4	Female	79.0	1	0	Yes	
5	Male	81.0	0	0	Yes	
6	Male	74.0	1	1	Yes	
7	Female	69.0	0	0	No	
10	Female	81.0	1	0	Yes	
11	Female	61.0	0	1	Yes	
12	Female	54.0	0	0	Yes	
14	Female	79.0	0	1	Yes	
15	Female	50.0	1	0	Yes	
16	Male	64.0	0	1	Yes	

Now, our data has been cleaned up, as we have removed the unhelpful ID column, factorized the categorical data, and removed rows containing NAs or unknown smoking status history

```
#USE dummy variable method; drop one of each set of columns created to avoid dummy variable trap;

### Categorical data to be converted to numeric data
categColumns = (["gender", "ever_married", "work_type", "Residence_type", "smoking_status"])

dummiesGender = pd.get_dummies(strokeData.gender)
dummiesMarried = pd.get_dummies(strokeData.ever_married)
dummiesWorkType = pd.get_dummies(strokeData.work_type)
dummiesResidence = pd.get_dummies(strokeData.Residence_type)
dummiesSmoking = pd.get_dummies(strokeData.smoking_status)


#Now add these to the original strokeData
merged = pd.concat([strokeData,dummiesGender],axis='columns') #remember there was a third option for 'other'
merged = pd.concat([merged,dummiesMarried],axis='columns') #remember no/yes column corresponds to married
merged = pd.concat([merged,dummiesWorkType],axis='columns')
merged = pd.concat([merged,dummiesResidence],axis='columns')
merged = pd.concat([merged,dummiesSmoking],axis='columns')

#Then remove the original categorical columns
mergedV2 = merged.drop(categColumns,axis='columns')

#Then remove one column from each new set of dummies from above to avoid the dummies trap
# numCol-1 is how many we will have for each
mergedV3 = mergedV2.drop(["Other", "Yes", "Self-employed", "Urban", "never smoked"],axis='columns')

#check
mergedV3 #21 to 16 columns;  this checks out since we dropped one for each categ column, which there are 5 of
```

	age	hypertension	heart_disease	avg_glucose_level	b
0	67.0	0	1	228.69	36
2	80.0	0	1	105.92	32
3	49.0	0	0	171.23	34
4	79.0	1	0	174.12	24
5	81.0	0	0	186.21	29
...	...	...	...	...	
5100	82.0	1	0	71.97	28
5102	57.0	0	0	77.93	21

```
#Now normalize the data for the non categorical values that have a much larger range than the rest
from sklearn.preprocessing import StandardScaler
cols_to_normalize = ['age', 'avg_glucose_level', 'bmi']
mergedV3[cols_to_normalize] = StandardScaler().fit_transform(mergedV3[cols_to_normalize])
mergedV3
```

	age	hypertension	heart_disease	avg_glucose_leve
0	0.973768	0	1	2.52362
2	1.663479	0	1	-0.05035
3	0.018784	0	0	1.31892
4	1.610424	1	0	1.37951
5	1.716533	0	0	1.63299
...	...	...	...	
5100	1.769588	1	0	-0.76214
5102	0.443222	0	0	-0.63719

```
#Split the majority and minority classes for the stroke column

df_minority = mergedV3[mergedV3['stroke'].astype(str).str.contains('1')]
print(len(df_minority))

#majority class of 0 for no stroke
df_majority= mergedV3[mergedV3['stroke'].astype(str).str.contains('0')]
print(len(df_majority))

#Now resample the minority class to match the majority class

df_minority_upsampled = resample(df_minority,
                                 replace=True,
                                 n_samples=len(df_majority),
                                 random_state=100)
df_balanced = pd.concat([df_majority, df_minority_upsampled])

#Now we have balanced the data
df_balanced['stroke'].value_counts()
print(df_balanced)
```

```
180
3246
          age  hypertension  heart_disease  avg_glucose_level       bmi  \
250  0.496276             1              0          -0.426905  1.221396
252  1.132932             0              0          -0.823579  0.769025
255  0.177948             0              0          -0.644321 -1.725871
256  1.398205             0              1           2.834755 -0.451007
257 -0.883145             0              0          -0.642643  0.275529
..        ...           ...            ...                ...       ...
180 -0.140380             0              0          -0.631951  0.069906
157  0.443222             0              0           2.381054  0.960940
97   0.496276             0              1           2.773115  0.152155
159  1.716533             1              0          -0.719169 -0.725171
122  1.663479             0              0           3.172305  0.193279

     stroke  Female  Male  No  Govt_job  Never_worked  Private  children  \
```

```
250      0      0    1   0         0            0       1         0
252      0      1    0   0         0            0       1         0
255      0      1    0   0         0            0       1         0
256      0      1    0   0         0            0       0         0
257      0      1    0   0         0            0       1         0
..     ...    ...  ...  ..       ...          ...     ...       ...
180      1      1    0   0         0            0       1         0
157      1      1    0   0         0            0       1         0
97       1      0    1   0         0            0       1         0
159      1      1    0   0         0            0       0         0
122      1      0    1   0         0            0       1         0

     Rural  formerly smoked  smokes
250      0                0       0
252      1                1       0
255      0                1       0
256      1                0       0
257      1                0       1
..     ...              ...     ...
180      0                0       0
157      0                0       1
97       1                0       1
159      1                0       0
122      1                0       1

[6492 rows x 16 columns]
```

```
#Now, lets split our data into our X and Y
#X should be our columns except for the stroke column
#Y should be our stroke column, our predicted output

#define by making x everything except dropped y column
# make y just the name of the last column
#split the data into features and target

X = df_balanced.drop('stroke', axis=1)
Y = df_balanced['stroke']


#Double check that this is the data for every column EXCEPT the target varaible 'stroke'
X
```

	age	hypertension	heart_disease	avg_glucose_level
250	0.496276	1	0	-0.426905

```
#Double check that this is the data for the Class Name/Target Variable 'stroke'
Y

print("And making sure there is a balanced number of each class")
print(df_balanced['stroke'].value_counts())
```

```
And making sure there is a balanced number of each class
0    3246
1    3246
Name: stroke, dtype: int64
```

```
#Now, lets split our data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X,Y,test_size=0.3, random_state=100, stratify=df_balanced['stroke'])
```

```
#Confirm the data was split appropriately
print(len(X_train))
print(len(X_test))
print(X.shape)
print(Y.shape)
print()
```

```
4544
1948
(6492, 15)
(6492,)
```

```
#Now check counts of label 0 and 1 before and after Over Sampling

print("TRAINING")
print("Before OverSampling, counts of label '1': {}".format(sum(y_train == 1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_train == 0)))

print("\nTESTING")
print("Before OverSampling, counts of label '1': {}".format(sum(y_test == 1)))
print("Before OverSampling, counts of label '0': {} \n".format(sum(y_test == 0)))

#The resampling to balance classes seems to have worked
```

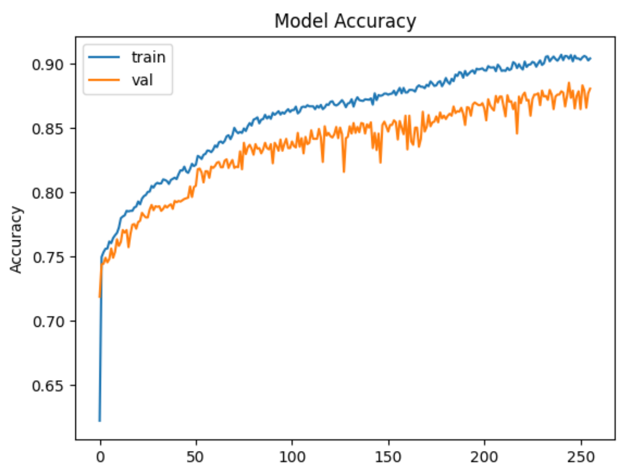
```
TRAINING
Before OverSampling, counts of label '1': 2272
Before OverSampling, counts of label '0': 2272


TESTING
Before OverSampling, counts of label '1': 974
Before OverSampling, counts of label '0': 974
```

```
#Show that having less neurons performs worse
#build the model
```

```
modelV3 = Sequential()
modelV3.add(Dense(15, activation='relu'))
modelV3.add(Dense(7, activation='relu'))
modelV3.add(Dense(1, activation='sigmoid'))

#compile the model
modelV3.compile(loss='binary_crossentropy',
                optimizer='adam', #standard optimizer
                metrics=['accuracy'])

#Train the model
historyV3 = modelV3.fit(x=X_train, y=y_train, epochs = 256, verbose=1, validation_data=(X_test, y_test))
```

```
Epoch 196/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2794 - accuracy: 0.8922 - val_loss: 0.3565 - val_accuracy: 0.8696
Epoch 197/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2810 - accuracy: 0.8959 - val_loss: 0.3498 - val_accuracy: 0.8686
Epoch 198/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2793 - accuracy: 0.8952 - val_loss: 0.3468 - val_accuracy: 0.8676
Epoch 199/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2808 - accuracy: 0.8957 - val_loss: 0.3456 - val_accuracy: 0.8676
```

```
#Graph for nn v3
#Check the learning curves
plt.plot(historyV3.history['accuracy'])
plt.plot(historyV3.history['val_accuracy'])

plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('epoch')
plt.legend(['train','val'],loc='upper left')
plt.show()
```




```
#NOW RUN ANOTHER MODEL BASED OFF OF https://www.youtube.com/watch?v=PM6uvCLyeXM

#build the model
modelV2 = Sequential()
keras.layers.Flatten(input_shape=(15,)),
modelV2.add(Dense(15, activation='relu'))
modelV2.add(Dense(15, activation='relu'))
modelV2.add(Dense(1, activation='sigmoid'))

#compile the model
modelV2.compile(loss='binary_crossentropy',
                optimizer='rmsprop', #standard optimizer
                metrics=['accuracy'])
```

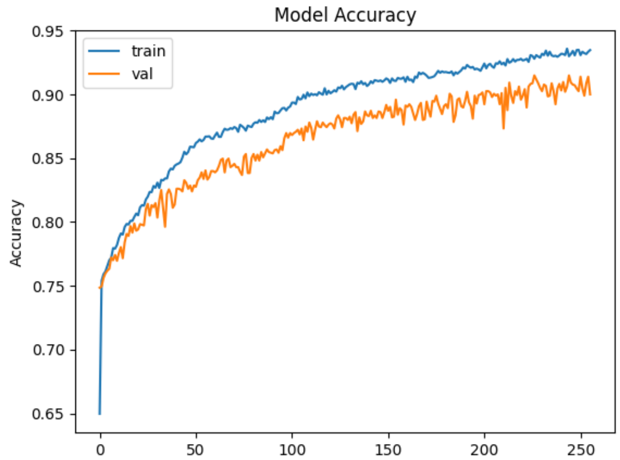
```
#Train the model
historyV2 = modelV2.fit(x=X_train, y=y_train, epochs = 256, verbose=1, validation_data=(X_test, y_test))
#historyV2 = modelV2.fit(X, Y, epochs = 256, verbose=1, validation_split=0.3)
```

```
Epoch 171/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2525 - accuracy: 0.9146 - val_loss: 0.3076 - val_accuracy: 0.8943
Epoch 172/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2501 - accuracy: 0.9129 - val_loss: 0.3109 - val_accuracy: 0.8876
Epoch 173/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2510 - accuracy: 0.9131 - val_loss: 0.3235 - val_accuracy: 0.8824
Epoch 174/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2500 - accuracy: 0.9135 - val_loss: 0.3082 - val_accuracy: 0.8948
Epoch 175/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2495 - accuracy: 0.9142 - val_loss: 0.3040 - val_accuracy: 0.8943
Epoch 176/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2509 - accuracy: 0.9162 - val_loss: 0.3050 - val_accuracy: 0.8989
Epoch 177/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2492 - accuracy: 0.9184 - val_loss: 0.3069 - val_accuracy: 0.8927
Epoch 178/256
142/142 [==============================] - 1s 4ms/step - loss: 0.2486 - accuracy: 0.9168 - val_loss: 0.3062 - val_accuracy: 0.8886
Epoch 179/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2485 - accuracy: 0.9179 - val_loss: 0.3178 - val_accuracy: 0.8845
Epoch 180/256
142/142 [==============================] - 0s 4ms/step - loss: 0.2492 - accuracy: 0.9173 - val_loss: 0.3099 - val_accuracy: 0.8907
Epoch 181/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2471 - accuracy: 0.9175 - val_loss: 0.3068 - val_accuracy: 0.8922
Epoch 182/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2462 - accuracy: 0.9170 - val_loss: 0.3107 - val_accuracy: 0.8891
Epoch 183/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2466 - accuracy: 0.9168 - val_loss: 0.3033 - val_accuracy: 0.9004
Epoch 184/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2460 - accuracy: 0.9179 - val_loss: 0.3037 - val_accuracy: 0.8984
Epoch 185/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2456 - accuracy: 0.9148 - val_loss: 0.3144 - val_accuracy: 0.8783
Epoch 186/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2461 - accuracy: 0.9173 - val_loss: 0.3075 - val_accuracy: 0.8835
Epoch 187/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2443 - accuracy: 0.9162 - val_loss: 0.3053 - val_accuracy: 0.8958
Epoch 188/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2452 - accuracy: 0.9175 - val_loss: 0.3048 - val_accuracy: 0.8968
Epoch 189/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2436 - accuracy: 0.9188 - val_loss: 0.3005 - val_accuracy: 0.8907
Epoch 190/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2434 - accuracy: 0.9210 - val_loss: 0.3124 - val_accuracy: 0.8896
Epoch 191/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2437 - accuracy: 0.9201 - val_loss: 0.3013 - val_accuracy: 0.8948
Epoch 192/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2431 - accuracy: 0.9230 - val_loss: 0.2999 - val_accuracy: 0.9020
Epoch 193/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2421 - accuracy: 0.9217 - val_loss: 0.3014 - val_accuracy: 0.8937
Epoch 194/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2426 - accuracy: 0.9208 - val_loss: 0.3053 - val_accuracy: 0.8907
Epoch 195/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2404 - accuracy: 0.9214 - val_loss: 0.3038 - val_accuracy: 0.8809
Epoch 196/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2432 - accuracy: 0.9195 - val_loss: 0.3046 - val_accuracy: 0.8989
Epoch 197/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2413 - accuracy: 0.9201 - val_loss: 0.3011 - val_accuracy: 0.8927
Epoch 198/256
142/142 [==============================] - 0s 3ms/step - loss: 0.2406 - accuracy: 0.9195 - val_loss: 0.3067 - val_accuracy: 0.8994
Epoch 199/256
142/142 [==============================] - 0s 2ms/step - loss: 0.2408 - accuracy: 0.9184 - val_loss: 0.3003 - val_accuracy: 0.8999
Epoch 200/256
```

```
#CAPTURE METRICS FOR THIS MODEL

prediction = modelV2.predict(X)
print("First few outputs; probability for first few datapoints")
print(prediction[:5])
print("\n")
print("First 5 values of predicted Y")
print(Y[:5])

#Now evaluate the model using scikit's accuracy metrics
my_accuracyV2 = accuracy_score(Y, prediction.round())
print("\nThe accuracy score is: ")
print(my_accuracyV2)

print("\nThe f1 score for this Neural Network is: ")
f1_score(Y, prediction.round())

#Cannot use the same Confusion Matrix Display on neural network like the other models.
```

```
203/203 [==============================] - 0s 1ms/step
First few outputs; probability for first few datapoints
[[1.5255224e-02]
 [3.4173377e-02]
 [1.6326828e-04]
 [4.5841026e-01]
 [5.5997580e-01]]


First 5 values of predicted Y
250    0
252    0
255    0
256    0
257    0
Name: stroke, dtype: int64

The accuracy score is:
0.916358595194085

The f1 score for this Neural Network is:
0.9218817436340093
```

```
#Graph for nn v2
#Check the learning curves
plt.plot(historyV2.history['accuracy'])
plt.plot(historyV2.history['val_accuracy'])

plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('epoch')
plt.legend(['train','val'],loc='upper left')
plt.show()
```




```
#Show more details about neural network model we ended up using
print(modelV2.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_6 (Dense)             (None, 15)                240

 dense_7 (Dense)             (None, 15)                240

 dense_8 (Dense)             (None, 1)                 16

=================================================================
Total params: 496
Trainable params: 496
Non-trainable params: 0
_________________________________________________________________
None
```

```
#Now lets generate the other metrics for our model such as precision, recall, and F1 scores

y_pred = modelV2.predict(X_test, batch_size=10, verbose=1)

print(classification_report(y_test, y_pred.round())) #use round to be able to determine 0 or 1 class since it predicts non integers
```

```
195/195 [==============================] - 0s 2ms/step
              precision    recall  f1-score   support

           0       0.98      0.82      0.89       974
           1       0.84      0.98      0.91       974

    accuracy                           0.90      1948
   macro avg       0.91      0.90      0.90      1948
weighted avg       0.91      0.90      0.90      1948
```

```
#Peek at what the unrounded predicted y_values are from the model
y_pred
```

```
array([[9.9990189e-01],
       [5.2611076e-06],
       [8.2818115e-01],
       ...,
       [9.8192418e-01],
       [9.9745905e-01],
       [9.8243916e-01]], dtype=float32)
```

Now, lets briefly run some other machine learning methods and take a quick look at their performance metrics. These will be done with the original unchanged x and y values from the first run of the neural network.

DECISION TREE CLASSIFIER

```
#DO HYPERPARAMETER TUNING ON DECISION TREE TO CHOOSE BEST PARAMETERS

#Define parameters
dt_parameters = {'criterion':('gini','entropy','log_loss'), 'splitter':('best','random')}

#Create a Decision Tree and GridSearch object to run the models
dt_hyp = DecisionTreeClassifier()
dt_hyp_gridSearch = GridSearchCV(dt_hyp, dt_parameters, scoring='accuracy')
dt_hyp_gridSearch.fit(X_train, y_train)

#Show results
dt_hyp_results = pd.DataFrame(dt_hyp_gridSearch.cv_results_)
dt_hyp_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_ti
0	0.021024	0.005348	0.003991	0.0005

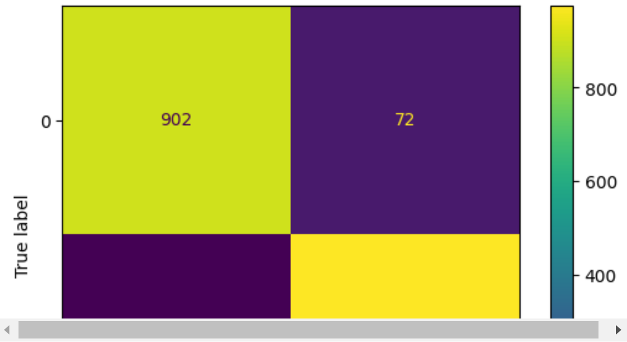
The parameters providing the #1 rank test score is using log_loss for criterion and 'random' for splitter

```
#Decision Tree Classifier

dc_clf = DecisionTreeClassifier(criterion='log_loss',splitter='random')
dc_clf.fit(X_train, y_train)
print("The score for the Decision Tree method is: ")
dc_clf.score(X_test, y_test)
```

```
#predict the target values for the test data
```

```
y_pred = dc_clf.predict(X_test)
#calculate accuracy of decision tree
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
#get classification report for decision tree
dt_report = classification_report(y_test, y_pred)
print(dt_report)

#Confusion Matrix
ConfusionMatrixDisplay.from_estimator(dc_clf, X_test, y_test)
```

```
The score for the Decision Tree method is:
Accuracy: 0.9630390143737166
              precision    recall  f1-score   support

           0       1.00      0.93      0.96       974
           1       0.93      1.00      0.96       974

    accuracy                           0.96      1948
   macro avg       0.97      0.96      0.96      1948
weighted avg       0.97      0.96      0.96      1948

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDispla
at 0x7f83dc3a41f0>
```




```
#DO HYPERPARAMETER TUNING ON RANDOM FOREST TO CHOOSE BEST PARAMETERS

#Define parameters
rf_parameters = {'criterion':('gini','entropy','log_loss'), 'oob_score':[True,False]}

#Create a Decision Tree and GridSearch object to run the models
rf_hyp = RandomForestClassifier()
rf_hyp_gridSearch = GridSearchCV(rf_hyp, rf_parameters, scoring='accuracy')
rf_hyp_gridSearch.fit(X_train, y_train)

#Show results
```

```
rf_hyp_results = pd.DataFrame(rf_hyp_gridSearch.cv_results_)
rf_hyp_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_ti
0	0.469976	0.088772	0.026908	0.0003

The parameters providing the #1 rank test score is using entropy for criterion and 'False' for oob_score

```
#Random Forest Classifier

rf_clf = RandomForestClassifier(criterion='entropy',oob_score=False)
rf_clf.fit(X_train, y_train)
print("The score for the Random Forest Classifier method is: ")
rf_clf.score(X_test, y_test)

#predict the target values for the test data
y_pred = rf_clf.predict(X_test)
#calculate accuracy of decision tree
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
#get classification report for decision tree
rf_report = classification_report(y_test, y_pred)
print(rf_report)

#Confusion Matrix
ConfusionMatrixDisplay.from_estimator(rf_clf, X_test, y_test)
```

```
The score for the Random Forest Classifier method is:
Accuracy: 0.9902464065708418
              precision    recall  f1-score   support

           0       1.00      0.98      0.99       974
           1       0.98      1.00      0.99       974

    accuracy                           0.99      1948
   macro avg       0.99      0.99      0.99      1948
weighted avg       0.99      0.99      0.99      1948

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDispla
at 0x7f83d44435e0>
```

```
#DO HYPERPARAMETER TUNING ON KNN TO CHOOSE BEST PARAMETERS

#Define parameters
knn_parameters = {'weights':('uniform','distance'), 'algorithm':('auto','ball_tree','kd_tree','brute')}

#Create a Decision Tree and GridSearch object to run the models
knn_hyp = KNeighborsClassifier()
knn_hyp_gridSearch = GridSearchCV(knn_hyp, knn_parameters, scoring='accuracy')
knn_hyp_gridSearch.fit(X_train, y_train)

#Show results
knn_hyp_results = pd.DataFrame(knn_hyp_gridSearch.cv_results_)
knn_hyp_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_ti
0	0.009337	0.001527	0.042893	0.0040
1	0.008551	0.001345	0.022038	0.0017

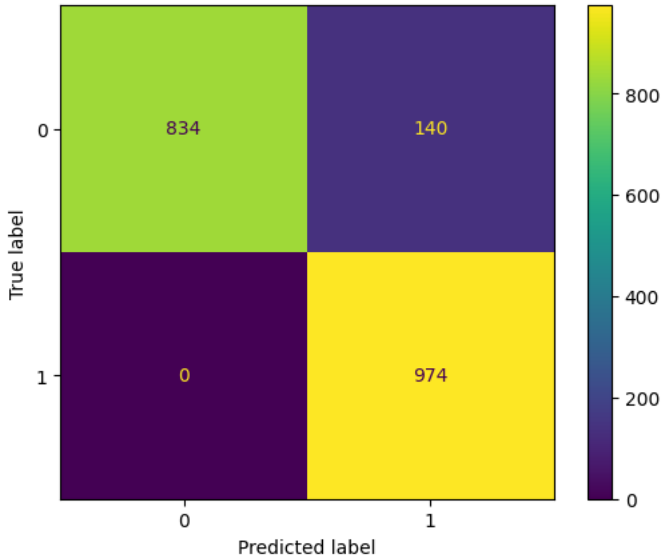
Several of the parameter setttings provided the same result, so we will pick the first one.

The chosen parameters with the highest rank order are 'auto' for algorithm and 'distance' for weights

```
#K-Nearest Neighbors (KNN)
knn_clf = KNeighborsClassifier(algorithm='auto', weights='distance')
knn_clf.fit(X_train, y_train)
print("The score for the K-Nearest Neighbors method is: ")
knn_clf.score(X_test, y_test)

#predict the target values for the test data
y_pred = knn_clf.predict(X_test)
```

```
#calculate accuracy of decision tree
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
#get classification report for decision tree
knn_report = classification_report(y_test, y_pred)
print(knn_report)

#Confusion Matrix
ConfusionMatrixDisplay.from_estimator(knn_clf, X_test, y_test)
```

```
The score for the K-Nearest Neighbors method is:
Accuracy: 0.9281314168377823
              precision    recall  f1-score   support

           0       1.00      0.86      0.92       974
           1       0.87      1.00      0.93       974

    accuracy                           0.93      1948
   macro avg       0.94      0.93      0.93      1948
weighted avg       0.94      0.93      0.93      1948

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f83d40c02b0>
```



```
#DO HYPERPARAMETER TUNING ON SVM TO CHOOSE BEST PARAMETERS

#Define parameters
sv_parameters = {'kernel':('linear','poly','rbf','sigmoid',), 'gamma':('auto','scale'), 'decision_function_shape':('ovo','ovr')}

#Create a Decision Tree and GridSearch object to run the models
sv_hyp = svm.SVC()
sv_hyp_gridSearch = GridSearchCV(sv_hyp, sv_parameters, scoring='accuracy')
sv_hyp_gridSearch.fit(X_train, y_train)

#Show results
sv_hyp_results = pd.DataFrame(sv_hyp_gridSearch.cv_results_)
sv_hyp_results
```

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_decision_function_shape	param_gamma	param_ke
0	0.655490	0.026304	0.069211	0.002787	ovo	auto	li
1	0.502074	0.123776	0.104221	0.032833	ovo	auto	
2	0.624212	0.176443	0.166625	0.041940	ovo	auto	
3	0.582563	0.075289	0.093957	0.001092	ovo	auto	sig
4	0.626925	0.016213	0.066249	0.001090	ovo	scale	li
5	0.671476	0.151073	0.105956	0.024087	ovo	scale	
6	0.488319	0.009658	0.121870	0.003455	ovo	scale	
7	0.468031	0.007236	0.103226	0.003832	ovo	scale	sig
8	0.626758	0.019053	0.066449	0.001302	ovr	auto	li
9	0.640864	0.127593	0.122614	0.035194	ovr	auto	

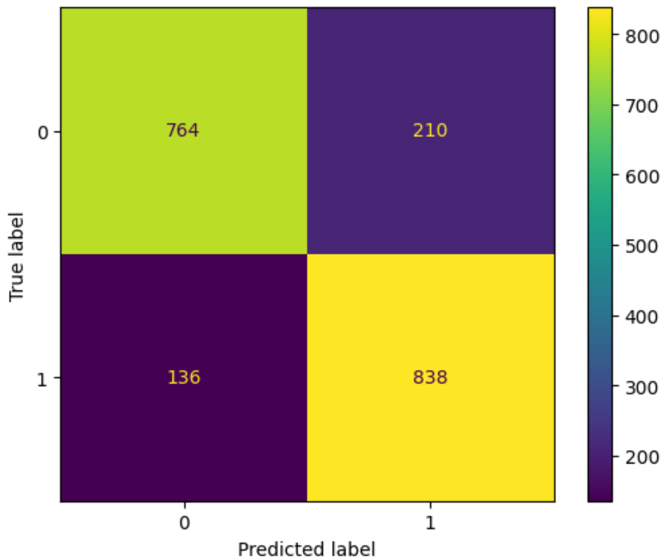
There are two results with the same rank order, so let's pick the first one.

The chosen parameters with the highest rank are: decision_function_shape=ovo, gamma=scale, and kernel=rbf

```
#Suppport Vector Machines (SVM)
sv_clf = svm.SVC(decision_function_shape='ovo',gamma='scale',kernel='rbf')
sv_clf.fit(X_train, y_train)
print("The score for the SVM method is: ")
sv_clf.score(X_test, y_test)

#predict the target values for the test data
y_pred = sv_clf.predict(X_test)
#calculate accuracy of decision tree
```

```
accuracy = accuracy_score(y_test, y_pred)
print('Accuracy:', accuracy)
#get classification report for decision tree
sv_report = classification_report(y_test, y_pred)
print(sv_report)

#Confusion Matrix
ConfusionMatrixDisplay.from_estimator(sv_clf, X_test, y_test)
```

```
The score for the SVM method is:
Accuracy: 0.8223819301848049
              precision    recall  f1-score   support

           0       0.85      0.78      0.82       974
           1       0.80      0.86      0.83       974

    accuracy                           0.82      1948
   macro avg       0.82      0.82      0.82      1948
weighted avg       0.82      0.82      0.82      1948

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f83d40a6ce0>
```



Colab paid products - Cancel contracts here

2s completed at 11:15 PM