```
def GEN(z, reuse=None):
    with tf.variiable_scope("", reuse = reuse):
        hid1 = tf.layers.dense(inputs = z, units = 128)
        alpha = 0.01
        hid1 = tf.maximum(alpha * hid1, hid1)
        hid2 = tf.layers.dense(inputs = hid1, units = 64)
        hid2 = tf.maximum(alpha * hid2, hid2)
        # change features
        outcome = tf.layers.dense(hid2, units = 14,
        activation = tf.nn.tanh)
        return outcome

def discriminator(X, reuse=None):
    with tf.variiable_scope("dis", reuse = reuse):
        hid1 = tf.layers.dense(inputs = X, units = 128)
        alpha = 0.01
        hid1 = tf.maximum(alpha * hid1, hid1)
        hid2 = tf.layers.dense(inputs = hid1, units = 128)
        hid2 = tf.maximum(alpha * hid2, hid2)
        logs = tf.layers.dense(hid2, units = 1)
        outcome = tf.sigmoid(logs)
        return outcome, logs

# the loss
def lossfunc(logs_in, labels_in):
    return tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logs(logs
    = logs_in, labels = labels_in))
```

```
def next_batch(num, data):
    idx = np.arange(0, len(data))
    np.random.shuffle(idx)
    idx = idx[:num]
    data_shuffle = [data.iloc[i] for i in idx]

    return np.asarray(data_shuffle)

real_data = tf.holder(tf.float32, shape = [None, 14]) # change features
z = tf.holder(tf.float32, shape = [None, 100])

G = GEN(z)

D_outcome_real, D_logs_real = discriminator(real_data)
D_outcome_fake, D_logs_fake = discriminator(G, reuse = True)

X_ones = X_train[] # choose your target column

realloss = lossfunc(D_logs_real, tf.ones_like(D_logs_real) * 0.9)
D_fake_loss = lossfunc(D_logs_fake, tf.zeros_like(D_logs_real))

lossD = realloss + D_fake_loss
lossG = lossfunc(D_logs_fake, tf.ones_like(D_logs_fake))

learnrate = 0.001
tvaris = tf.trainable_variiables()
```

```
dvaris = [vari for vari in tvaris if 'dis' in vari.name]
g_varis = [vari for vari in tvaris if '' in vari.name]

Dtrain = tf.train.AdamOptimizer(learnrate
= learnrate).minimize(lossD, vari_list = dvaris)
Gtrain = tf.train.AdamOptimizer(learnrate
= learnrate).minimize(lossG, vari_list = g_varis)

X_ones.reset_index(drop = True, inplace = True)

batch_size = 105
epochs = 500
init = tf.global_variiables_initializer()
samples = []
no_of_samples_to_erate = len(y_train[]) - len(X_ones)
print(len(y_train[]) - len(X_ones))

with tf.Session() as sess:
    sess.run(init)

    for epoch in range(epochs):

        num_batches = len(X_ones) // batch_size
        for i in range(num_batches):

            batch_X = next_batch(batch_size, X_ones)
```

```
            batch_z = np.random.uniform(-1,1, size = (batch_size,100))

            _ = sess.run(Dtrain, feed_dict = {real_data:batch_X,
            z: batch_z})
            _ = sess.run(Gtrain, feed_dict = {z: batch_z})

        if epoch % 100 == 0:
            print("ON Epoch {}".format(epoch))

    # new samples will be generated
    sampleZ = np.random.uniform(-1,1,size = (no_of_samples_to_erate, 100))
    _Sample = sess.run(erator(z, reuse = True), feed_dict = {z:sampleZ})
    samples.append(_Sample)

X_fake = pd.DataFrame(samples[0], columns = X.columns)
```

0.1 SMOTE-based Approaches

```
# SMOTE
sos = SMOTE(random_state = 0)
X_sos, y_sos = sos.fit_sample(X_train, y_train)

# SMOTE-Tomek
kos = SMOTETomek(random_state = 0)
X_kos, y_kos = kos.fit_sample(X_train, y_train)
```

```
# SMOTE-ENN
kos = SMOTETomek(random_state = 0)
X_kos, y_kos = kos.fit_sample(X_train, y_train)
```

0.2 Classification Models

```
models = []
models.append(('KNN', KNeighborsClassifier(n_neighbors = 10,leaf_size = 30)))
models.append(('RF', RandomForestClassifier()))
models.append(('XGB',xgb.XGBClassifier(objective = "binary:logistic",
random_state = 2,max_depth=8,n_estimators = 50)))

def evaluate_model(clf, X, y):

    pred = clf.predict(X) # predicted classes
    accuracy = accuracy_score(pred,y) # calculate accuracy
    report = classification_report(y, pred, labels = [], outcome_dict = True)
    report_df = pd.DataFrame(report).transpose()
    report_df = report_df.reset_index()
    model_eval = report_df[report_df['index'].str.contains('1')][['precision',
    'recall', 'f1-score']]
    cf_matrix = confusion_matrix(y, pred)

    return model_eval

names = []
scores = []
```

```
for name, model in models:
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    scores.append(sklearn.metrics.f1_score(y_test, y_pred,
    average = 'weighted'))
    names.append(name)
tr_split = pd.DataFrame({'Name': names, 'Score': scores})
print(tr_split)
```