Assignments 4

November 17, 2019

1 MScBMI 33200 – Machine Learning for Biomedical Informatics

2 Assignment IV

Name: Troy Zhongyi Zhang
Netid: zhongyiz@uchicago.edu

2.0.1 Directions:

1. Fill out below information (tables and methods)
2. Submit this document along with your code in an HTML/PDF format

2.0.2 Section 1: EMR Bots 30-day Readmission study

Using the training datasets, create the following models: 1. Naïve model: This model utilizes only patient characteristics (age, gender and race) to predict 30-day readmission in a logistic regression framework

2. GLM model : This model utilizes patient characteristics and most-recent lab recordings to predict 30-day admissions in a logistic regression framework.

3. ANN model: This model utilizes patient characteristics and most-recent lab recordings to predict 30-day admissions using an artificial neural network. Feature engineering steps include balancing classes using SMOTE as well as data normalization/standardization of continuous variables.

4. RF Model : This model utilizes patient characteristics and most-recent lab recordings to predict 30-day admissions using a random forest.

5. GBM Model: This model utilizes patient characteristics and most-recent lab recordings to predict 30-day admissions using a gradient boosted machine. Utilize a five-fold cross-validation technique to build your model. Calculate AUC on the test dataset. Fill out the following Table.

[1]:
```
import pandas as pd
import numpy as np
from random import seed
seed(1)
```

[2]:
```
r_outcome = pd.read_csv("readmission_outcome.csv")
info = pd.read_csv("encounter_info.csv")
#df1 = pd.merge(info, r_outcome, on = "Encounter_ID")
labs = pd.read_csv("encounter_labs.csv")
```

[3]:
```
labs = labs.groupby(['Encounter_ID']).tail(1)
labs = labs.reset_index(drop=True)
wrong_ids = labs.iloc[:7]
wrong_ids['Encounter_ID'] = pd.DataFrame(wrong_ids['Encounter_ID'
                                                    ]).applymap(lambda x: x.
 ↪replace('1e+05', '100000'))
labs.head()
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:5:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """
```

[3]:
```
  Encounter_ID             Lab_DTTM  CBC..ABSOLUTE.LYMPHOCYTES  \
0     100000_1  1981-01-08 23:28:24                       32.8
1     100000_2  1996-02-06 17:07:30                       16.1
2     100000_3  2002-04-11 00:36:27                       23.4
3     100000_4  2006-11-29 12:01:11                       15.7
4     100000_5  2007-04-27 16:49:40                       27.4

   CBC..ABSOLUTE.NEUTROPHILS  CBC..BASOPHILS  CBC..EOSINOPHILS  \
0                       76.4             0.0               0.1
1                       77.9             0.2               0.4
2                       66.2             0.1               0.4
3                       78.3             0.0               0.2
4                       62.2             0.1               0.4

   CBC..HEMATOCRIT  CBC..HEMOGLOBIN  CBC..PLATELET.COUNT  \
0             35.9             14.8                386.0
1             49.6             16.2                339.2
2             40.4             18.7                408.7
3             54.4             18.9                128.6
4             45.2             16.8                315.4

   CBC..RED.BLOOD.CELL.COUNT  CBC..WHITE.BLOOD.CELL.COUNT  METABOLIC..ALBUMIN  \
0                        4.8                         10.7                 5.8
1                        5.8                          6.4                 2.7
2                        3.2                          8.5                 3.7
```

```
3                        5.9                           7.0                 4.0
4                        4.9                          11.8                 5.9

   METABOLIC..BILI.TOTAL  METABOLIC..BUN  METABOLIC..CALCIUM  \
0                    0.8            12.6                11.1
1                    0.9            15.6                 8.1
2                    0.3            15.1                 7.9
3                    0.9            24.4                 7.0
4                    0.1            21.4                 7.5

   METABOLIC..CREATININE  METABOLIC..POTASSIUM  METABOLIC..SODIUM
0                    0.9                   4.1              135.5
1                    1.2                   4.7              140.9
2                    0.7                   5.3              150.0
3                    1.1                   4.4              155.0
4                    0.8                   5.6              137.1
```

[4]:

```
read1 = pd.merge(info, labs, on = "Encounter_ID")
read = pd.merge(read1,r_outcome, on = "Encounter_ID")
read['PatientGender'] = read['PatientGender'].replace('Female',1)
read['PatientGender'] = read['PatientGender'].replace('Male',0)
read['PatientGender'] = read['PatientGender'].astype('category').cat.codes

read['PatientRace'] = read['PatientRace'].replace('African American',0)
read['PatientRace'] = read['PatientRace'].replace('White',1)
read['PatientRace'] = read['PatientRace'].replace('Asian',2)
read['PatientRace'] = read['PatientRace'].replace('Unknown',3)
read['PatientRace'] = read['PatientRace'].astype('category').cat.codes

read_train = read[read["AdmissionEndDate"].str[:4].astype(int)<=2004]
read_test = read[read["AdmissionEndDate"].str[:4].astype(int)>2004]
# read_train = read_train.reset_index(drop=True)
# read_test = read_test.reset_index(drop=True)

read_Xtrain1 = read_train[['PatientEncounterAge','PatientGender','PatientRace']]
read_ytrain1 = read_train[['outcome']]
read_Xtest1 = read_test[['PatientEncounterAge','PatientGender','PatientRace']]
read_ytest1 = read_test[['outcome']]
```

3 Naïve model:

3.0.1 readmission - read

[5]:
```
from sklearn.model_selection import GridSearchCV, StratifiedKFold,␣
↪train_test_split
from sklearn.naive_bayes import GaussianNB
from random import seed
seed(0)
# skf = StratifiedKFold(n_splits=5)
params1 = {'var_smoothing' : [1e-10,1e-9,1e-7,1e-5,1e-3]}
nb = GaussianNB()
gs1 = GridSearchCV(nb, cv=5, param_grid=params1, scoring = 'roc_auc',refit =␣
↪True, n_jobs=-1, verbose = 5, return_train_score=True)

gs1.fit(read_Xtrain1, read_ytrain1)
gs1.cv_results_
```

```
Fitting 5 folds for each of 5 candidates, totalling 25 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done   8 out of  25 | elapsed:    2.1s remaining:    4.5s
[Parallel(n_jobs=-1)]: Done  14 out of  25 | elapsed:    2.2s remaining:    1.7s
[Parallel(n_jobs=-1)]: Done  20 out of  25 | elapsed:    2.3s remaining:    0.6s
[Parallel(n_jobs=-1)]: Done  25 out of  25 | elapsed:    2.3s finished
/usr/local/lib/python3.7/site-packages/sklearn/utils/validation.py:724:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

[5]:
```
{'mean_fit_time': array([0.01098027, 0.01187506, 0.01067371, 0.01098475,
0.00743365]),
 'std_fit_time': array([0.00046294, 0.00205628, 0.00064158, 0.00066528,
0.00163095]),
 'mean_score_time': array([0.00680275, 0.00860696, 0.00756755, 0.00757775,
0.00481987]),
 'std_score_time': array([0.00033094, 0.00157957, 0.00108967, 0.00098609,
0.00071321]),
 'param_var_smoothing': masked_array(data=[1e-10, 1e-09, 1e-07, 1e-05, 0.001],
              mask=[False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'var_smoothing': 1e-10},
  {'var_smoothing': 1e-09},
  {'var_smoothing': 1e-07},
  {'var_smoothing': 1e-05},
```

```
 {'var_smoothing': 0.001}],
 'split0_test_score': array([0.4152807 , 0.4152807 , 0.41529528, 0.4155433 ,
0.42218137]),
 'split1_test_score': array([0.53448226, 0.53448226, 0.53448226, 0.53470114,
0.55580201]),
 'split2_test_score': array([0.5161248 , 0.5161248 , 0.5161248 , 0.51625613,
0.51646042]),
 'split3_test_score': array([0.72176823, 0.72176823, 0.72176823, 0.72172153,
0.72019612]),
 'split4_test_score': array([0.69091758, 0.69091758, 0.69091758, 0.69102654,
0.68845825]),
 'mean_test_score': array([0.57569509, 0.57569509, 0.57569801, 0.57583013,
0.58060075]),
 'std_test_score': array([0.11453774, 0.11453774, 0.11453365, 0.11444477,
0.1104058 ]),
 'rank_test_score': array([4, 4, 3, 2, 1], dtype=int32),
 'split0_train_score': array([0.62274145, 0.62274145, 0.62274239, 0.62273297,
0.62181505]),
 'split1_train_score': array([0.5788951 , 0.5788951 , 0.57889415, 0.57892522,
0.58229167]),
 'split2_train_score': array([0.59410438, 0.59410438, 0.59410532, 0.59421829,
0.60132493]),
 'split3_train_score': array([0.55049987, 0.55049987, 0.55049987, 0.55052211,
0.55243154]),
 'split4_train_score': array([0.56591058, 0.56591058, 0.56591151, 0.56609588,
0.56631545]),
 'mean_train_score': array([0.58243028, 0.58243028, 0.58243065, 0.58249889,
0.58483573]),
 'std_train_score': array([0.02476378, 0.02476378, 0.02476408, 0.02474054,
0.02465307])}
```

[6]:
```
gs1.best_params_
```

[6]:
```
{'var_smoothing': 0.001}
```

[7]:
```
gs1.best_estimator_
```

[7]:
```
GaussianNB(priors=None, var_smoothing=0.001)
```

[8]:
```
y_pred1 = gs1.best_estimator_.predict(read_Xtest1)
y_pred1
```

[8]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[9]:
```
prob1 = gs1.best_estimator_.predict_proba(read_Xtest1)
prob1
```

[9]:
```
array([[0.99597631, 0.00402369],
       [0.99524932, 0.00475068],
       [0.99602356, 0.00397644],
       …,
       [0.98229542, 0.01770458],
       [0.99532479, 0.00467521],
       [0.99431079, 0.00568921]])
```

[10]:
```
from sklearn import metrics
nb_matrix1 = metrics.confusion_matrix(read_ytest1, y_pred1)
nb_matrix1
```

[10]:
```
array([[14599,     0],
       [   50,     0]])
```

[11]:
```
target_names1 = ['Not in 30 days', 'Readmitted within 30 days']
from sklearn.metrics import classification_report
print("", classification_report(read_ytest1, y_pred1,␣
 ↪target_names=target_names1))
```

```
                           precision    recall  f1-score   support

           Not in 30 days       1.00      1.00      1.00     14599
Readmitted within 30 days       0.00      0.00      0.00        50

                 accuracy                           1.00     14649
                macro avg       0.50      0.50      0.50     14649
             weighted avg       0.99      1.00      0.99     14649

/usr/local/lib/python3.7/site-packages/sklearn/metrics/classification.py:1437:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples.
  'precision', 'predicted', average, warn_for)
```

[12]:
```
from sklearn.metrics import roc_auc_score
nb_probs = gs1.best_estimator_.predict_proba(read_Xtest1)[:,1]
print(roc_auc_score(read_ytest1, nb_probs))
```

```
0.4905582574148914
```

[13]:
```
from sklearn.metrics import accuracy_score
as1 = accuracy_score(read_ytest1, y_pred1)
```

[14]:
```
as1
```

[14]: 0.9965867977336337

[15]:
```python
error1 = 1-as1
error1
```

[15]: 0.003413202266366322

[16]:
```python
n1 = len(y_pred1)
```

[17]:
```python
import math
error1 + 1.96 * math.sqrt((error1 * (1 - error1)) / n1)
```

[17]: 0.004357677685114603

[18]:
```python
error1 - 1.96 * math.sqrt((error1 * (1 - error1)) / n1)
```

[18]: 0.0024687268476180405

[19]:
```python
import numpy as np, scipy.stats as st
st.t.interval(0.95, len(read_ytest1)-1, loc=np.mean(read_ytest1), scale=st.
 ↪sem(read_ytest1))
```

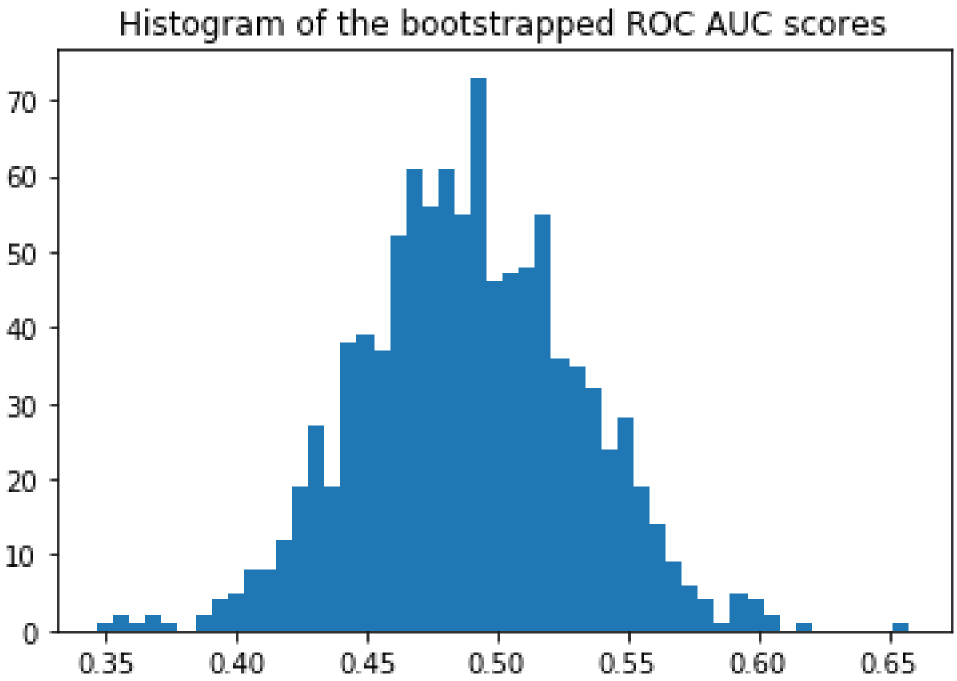
[19]: (array([0.00246863]), array([0.00435777]))

[20]:
```python
read_ytest1 = read_ytest1.values
```

[22]:
```python
#Calculated the Confidence Interval by bootstrapping
import numpy as np
from scipy.stats import sem
from sklearn.metrics import roc_auc_score

y_pred = nb_probs
y_true = read_ytest1

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue
```

```python
    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.4906
```




```
Confidence interval for the score: [0.4232 - 0.5571]
```

[23]:
```
#Transplanted the pROC package from R into Python for CI computation
import numpy as np
import scipy.stats
from scipy import stats

read_ytest1=read_ytest1.reshape((14649,))
# AUC comparison adapted from
# https://github.com/Netflix/vmaf/
def compute_midrank(x):
    """Computes midranks.
    Args:
       x - a 1D numpy array
    Returns:
       array of midranks
    """
    J = np.argsort(x)
    Z = x[J]
    N = len(x)
    T = np.zeros(N, dtype=np.float)
    i = 0
    while i < N:
        j = i
        while j < N and Z[j] == Z[i]:
            j += 1
        T[i:j] = 0.5*(i + j - 1)
        i = j
    T2 = np.empty(N, dtype=np.float)
    # Note(kazeevn) +1 is due to Python using 0-based indexing
    # instead of 1-based in the AUC formula in the paper
    T2[J] = T + 1
    return T2


def compute_midrank_weight(x, sample_weight):
    """Computes midranks.
    Args:
       x - a 1D numpy array
    Returns:
       array of midranks
    """
    J = np.argsort(x)
    Z = x[J]
    cumulative_weight = np.cumsum(sample_weight[J])
    N = len(x)
    T = np.zeros(N, dtype=np.float)
    i = 0
    while i < N:
```

```
        j = i
        while j < N and Z[j] == Z[i]:
            j += 1
        T[i:j] = cumulative_weight[i:j].mean()
        i = j
    T2 = np.empty(N, dtype=np.float)
    T2[J] = T
    return T2


def fastDeLong(predictions_sorted_transposed, label_1_count, sample_weight):
    if sample_weight is None:
        return fastDeLong_no_weights(predictions_sorted_transposed,␣
 ↪label_1_count)
    else:
        return fastDeLong_weights(predictions_sorted_transposed, label_1_count,␣
 ↪sample_weight)


def fastDeLong_weights(predictions_sorted_transposed, label_1_count,␣
 ↪sample_weight):
    """
    The fast version of DeLong's method for computing the covariance of
    unadjusted AUC.
    Args:
       predictions_sorted_transposed: a 2D numpy.array[n_classifiers,␣
 ↪n_examples]
          sorted such as the examples with label "1" are first
    Returns:
       (AUC value, DeLong covariance)
    Reference:
     @article{sun2014fast,
       title={Fast Implementation of DeLong's Algorithm for
              Comparing the Areas Under Correlated Receiver Oerating␣
 ↪Characteristic Curves},
       author={Xu Sun and Weichao Xu},
       journal={IEEE Signal Processing Letters},
       volume={21},
       number={11},
       pages={1389--1393},
       year={2014},
       publisher={IEEE}
     }
    """
    # Short variables are named as they are in the paper
    m = label_1_count
    n = predictions_sorted_transposed.shape[1] - m
```

```
    positive_examples = predictions_sorted_transposed[:, :m]
    negative_examples = predictions_sorted_transposed[:, m:]
    k = predictions_sorted_transposed.shape[0]

    tx = np.empty([k, m], dtype=np.float)
    ty = np.empty([k, n], dtype=np.float)
    tz = np.empty([k, m + n], dtype=np.float)
    for r in range(k):
        tx[r, :] = compute_midrank_weight(positive_examples[r, :],␣
↪sample_weight[:m])
        ty[r, :] = compute_midrank_weight(negative_examples[r, :],␣
↪sample_weight[m:])
        tz[r, :] = compute_midrank_weight(predictions_sorted_transposed[r, :],␣
↪sample_weight)
    total_positive_weights = sample_weight[:m].sum()
    total_negative_weights = sample_weight[m:].sum()
    pair_weights = np.dot(sample_weight[:m, np.newaxis], sample_weight[np.
↪newaxis, m:])
    total_pair_weights = pair_weights.sum()
    aucs = (sample_weight[:m]*(tz[:, :m] - tx)).sum(axis=1) / total_pair_weights
    v01 = (tz[:, :m] - tx[:, :]) / total_negative_weights
    v10 = 1. - (tz[:, m:] - ty[:, :]) / total_positive_weights
    sx = np.cov(v01)
    sy = np.cov(v10)
    delongcov = sx / m + sy / n
    return aucs, delongcov


def fastDeLong_no_weights(predictions_sorted_transposed, label_1_count):
    """
    The fast version of DeLong's method for computing the covariance of
    unadjusted AUC.
    Args:
       predictions_sorted_transposed: a 2D numpy.array[n_classifiers,␣
↪n_examples]
          sorted such as the examples with label "1" are first
    Returns:
       (AUC value, DeLong covariance)
    Reference:
     @article{sun2014fast,
       title={Fast Implementation of DeLong's Algorithm for
              Comparing the Areas Under Correlated Receiver Oerating
              Characteristic Curves},
       author={Xu Sun and Weichao Xu},
       journal={IEEE Signal Processing Letters},
       volume={21},
       number={11},
```

```
      pages={1389--1393},
      year={2014},
      publisher={IEEE}
    }
    """
    # Short variables are named as they are in the paper
    m = label_1_count
    n = predictions_sorted_transposed.shape[1] - m
    positive_examples = predictions_sorted_transposed[:, :m]
    negative_examples = predictions_sorted_transposed[:, m:]
    k = predictions_sorted_transposed.shape[0]

    tx = np.empty([k, m], dtype=np.float)
    ty = np.empty([k, n], dtype=np.float)
    tz = np.empty([k, m + n], dtype=np.float)
    for r in range(k):
        tx[r, :] = compute_midrank(positive_examples[r, :])
        ty[r, :] = compute_midrank(negative_examples[r, :])
        tz[r, :] = compute_midrank(predictions_sorted_transposed[r, :])
    aucs = tz[:, :m].sum(axis=1) / m / n - float(m + 1.0) / 2.0 / n
    v01 = (tz[:, :m] - tx[:, :]) / n
    v10 = 1.0 - (tz[:, m:] - ty[:, :]) / m
    sx = np.cov(v01)
    sy = np.cov(v10)
    delongcov = sx / m + sy / n
    return aucs, delongcov


def calc_pvalue(aucs, sigma):
    """Computes log(10) of p-values.
    Args:
       aucs: 1D array of AUCs
       sigma: AUC DeLong covariances
    Returns:
       log10(pvalue)
    """
    l = np.array([[1, -1]])
    z = np.abs(np.diff(aucs)) / np.sqrt(np.dot(np.dot(l, sigma), l.T))
    return np.log10(2) + scipy.stats.norm.logsf(z, loc=0, scale=1) / np.log(10)


def compute_ground_truth_statistics(ground_truth, sample_weight):
    assert np.array_equal(np.unique(ground_truth), [0, 1])
    order = (-ground_truth).argsort()
    label_1_count = int(ground_truth.sum())
    if sample_weight is None:
        ordered_sample_weight = None
```

```python
    else:
        ordered_sample_weight = sample_weight[order]

    return order, label_1_count, ordered_sample_weight


def delong_roc_variance(ground_truth, predictions, sample_weight=None):
    """
    Computes ROC AUC variance for a single set of predictions
    Args:
       ground_truth: np.array of 0 and 1
       predictions: np.array of floats of the probability of being class 1
    """
    order, label_1_count, ordered_sample_weight =␣
 ↪compute_ground_truth_statistics(
        ground_truth, sample_weight)
    predictions_sorted_transposed = predictions[np.newaxis, order]
    aucs, delongcov = fastDeLong(predictions_sorted_transposed, label_1_count,␣
 ↪ordered_sample_weight)
    assert len(aucs) == 1, "There is a bug in the code, please forward this to␣
 ↪the developers"
    return aucs[0], delongcov


alpha = .95
y_pred = nb_probs
y_true = read_ytest1

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.4905582574148914
```

```
AUC COV: 0.0016752140380696195
95% AUC CI: [0.41033815 0.57077837]
```

4 Logistic Regression model

[24]:
```python
read_Xtrain2 = read_train.
↪drop(["Patient_ID","Encounter_ID","AdmissionStartDate",
                                "AdmissionEndDate","Lab_DTTM",␣
↪"outcome"],axis=1)
read_ytrain2 = read_train[['outcome']]
read_Xtest2 = read_test.drop(["Patient_ID","Encounter_ID","AdmissionStartDate",
                              "AdmissionEndDate","Lab_DTTM", "outcome"],axis=1)
read_ytest2 = read_test[['outcome']]
```

[25]:
```python
from sklearn.linear_model import LogisticRegression
seed(0)
# skf = StratifiedKFold(n_splits=5)
params2 = {'tol' : [1e-6,1e-5,1e-4,1e-3,1e-2],
           'C': [0.5,1.0,1.5,2.0,2.5]}
lg1 = LogisticRegression(random_state=0,␣
↪solver='lbfgs',multi_class='multinomial')
lg1 = GridSearchCV(lg1, cv=5, param_grid=params2, scoring = 'roc_auc',refit =␣
↪True,
                   n_jobs=-1, verbose = 5, return_train_score=True)

lg1.fit(read_Xtrain2, read_ytrain2)
lg1.cv_results_
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:    6.8s
[Parallel(n_jobs=-1)]: Done 125 out of 125 | elapsed:   13.5s finished
/usr/local/lib/python3.7/site-packages/sklearn/utils/validation.py:724:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
/usr/local/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:947:
ConvergenceWarning: lbfgs failed to converge. Increase the number of iterations.
  "of iterations.", ConvergenceWarning)
```

[25]:
```
{'mean_fit_time': array([0.99134631, 1.05234489, 1.1355967 , 1.14954338,
1.11083565,
        1.13382444, 1.11382279, 1.10081606, 1.08704262, 1.05061502,
        1.05289245, 1.03408618, 1.0322576 , 1.02176347, 1.01946154,
```

```
       1.01821108, 1.02607713, 1.0263484 , 1.01205664, 1.04532151,
       1.04215569, 1.02043476, 1.00534463, 0.9680397 , 0.53218379]),
 'std_fit_time': array([0.05856146, 0.06390946, 0.0783923 , 0.048955  ,
0.05379491,
       0.07302304, 0.06837048, 0.05893198, 0.07598414, 0.05214384,
       0.06903119, 0.06669718, 0.06321412, 0.06051287, 0.06429707,
       0.04398919, 0.0460695 , 0.03534384, 0.04906314, 0.05816847,
       0.05056894, 0.05300128, 0.05234357, 0.05537495, 0.06439038]),
 'mean_score_time': array([0.01269679, 0.0134213 , 0.01065922, 0.00874157,
0.01034584,
       0.00917377, 0.00964613, 0.00910587, 0.009622  , 0.0093102 ,
       0.00935454, 0.0103436 , 0.01124744, 0.00921335, 0.00927849,
       0.00934606, 0.00947814, 0.00928826, 0.00955462, 0.01034303,
       0.00884237, 0.00965886, 0.01023545, 0.00668058, 0.00407419]),
 'std_score_time': array([0.00331538, 0.00173029, 0.00196584, 0.00012994,
0.0014411 ,
       0.00100156, 0.00085938, 0.00056496, 0.00125228, 0.00026267,
       0.00059344, 0.00161009, 0.00397683, 0.00059141, 0.00053022,
       0.00091892, 0.00093374, 0.00080137, 0.00113425, 0.00250453,
       0.0003655 , 0.0005053 , 0.0008346 , 0.00146997, 0.00029178]),
 'param_C': masked_array(data=[0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0,
1.5,
                   1.5, 1.5, 1.5, 1.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.5, 2.5,
                   2.5, 2.5, 2.5],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False],
       fill_value='?',
            dtype=object),
 'param_tol': masked_array(data=[1e-06, 1e-05, 0.0001, 0.001, 0.01, 1e-06,
1e-05,
                   0.0001, 0.001, 0.01, 1e-06, 1e-05, 0.0001, 0.001, 0.01,
                   1e-06, 1e-05, 0.0001, 0.001, 0.01, 1e-06, 1e-05,
                   0.0001, 0.001, 0.01],
             mask=[False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False, False, False, False, False, False, False, False,
                   False],
       fill_value='?',
            dtype=object),
 'params': [{'C': 0.5, 'tol': 1e-06},
  {'C': 0.5, 'tol': 1e-05},
  {'C': 0.5, 'tol': 0.0001},
  {'C': 0.5, 'tol': 0.001},
  {'C': 0.5, 'tol': 0.01},
  {'C': 1.0, 'tol': 1e-06},
```

```
 {'C': 1.0, 'tol': 1e-05},
 {'C': 1.0, 'tol': 0.0001},
 {'C': 1.0, 'tol': 0.001},
 {'C': 1.0, 'tol': 0.01},
 {'C': 1.5, 'tol': 1e-06},
 {'C': 1.5, 'tol': 1e-05},
 {'C': 1.5, 'tol': 0.0001},
 {'C': 1.5, 'tol': 0.001},
 {'C': 1.5, 'tol': 0.01},
 {'C': 2.0, 'tol': 1e-06},
 {'C': 2.0, 'tol': 1e-05},
 {'C': 2.0, 'tol': 0.0001},
 {'C': 2.0, 'tol': 0.001},
 {'C': 2.0, 'tol': 0.01},
 {'C': 2.5, 'tol': 1e-06},
 {'C': 2.5, 'tol': 1e-05},
 {'C': 2.5, 'tol': 0.0001},
 {'C': 2.5, 'tol': 0.001},
 {'C': 2.5, 'tol': 0.01}],
'split0_test_score': array([0.60133637, 0.60133637, 0.60133637, 0.60133637,
0.60133637,
       0.60673436, 0.60673436, 0.60673436, 0.60673436, 0.60673436,
       0.60903945, 0.60903945, 0.60903945, 0.60903945, 0.60903945,
       0.59485878, 0.59485878, 0.59485878, 0.59485878, 0.59485878,
       0.60566935, 0.60566935, 0.60566935, 0.60566935, 0.60566935]),
'split1_test_score': array([0.51786131, 0.51786131, 0.51786131, 0.51786131,
0.51786131,
       0.51702954, 0.51702954, 0.51702954, 0.51702954, 0.51702954,
       0.52021072, 0.52021072, 0.52021072, 0.52021072, 0.52021072,
       0.51958324, 0.51958324, 0.51958324, 0.51958324, 0.51958324,
       0.51845961, 0.51845961, 0.51845961, 0.51845961, 0.51845961]),
'split2_test_score': array([0.46472975, 0.46472975, 0.46472975, 0.46472975,
0.46472975,
       0.44697058, 0.44697058, 0.44697058, 0.44697058, 0.44697058,
       0.44176103, 0.44176103, 0.44176103, 0.44176103, 0.44176103,
       0.45257413, 0.45257413, 0.45257413, 0.45257413, 0.45257413,
       0.44186318, 0.44186318, 0.44186318, 0.44186318, 0.44186318]),
'split3_test_score': array([0.59735388, 0.59735388, 0.59735388, 0.59735388,
0.59735388,
       0.59321348, 0.59321348, 0.59321348, 0.59321348, 0.59321348,
       0.59736945, 0.59736945, 0.59736945, 0.59736945, 0.59736945,
       0.59234182, 0.59234182, 0.59234182, 0.59234182, 0.59234182,
       0.59394505, 0.59394505, 0.59394505, 0.59394505, 0.59394505]),
'split4_test_score': array([0.66300879, 0.66300879, 0.66300879, 0.66300879,
0.66300879,
       0.66825434, 0.66825434, 0.66825434, 0.66825434, 0.66825434,
       0.66319558, 0.66319558, 0.66319558, 0.66319558, 0.66319558,
```

```
        0.66682232, 0.66682232, 0.66682232, 0.66682232, 0.66682232,
        0.66302436, 0.66302436, 0.66302436, 0.66302436, 0.66302436]),
 'mean_test_score': array([0.56885382, 0.56885382, 0.56885382, 0.56885382,
0.56885382,
        0.56643635, 0.56643635, 0.56643635, 0.56643635, 0.56643635,
        0.56631128, 0.56631128, 0.56631128, 0.56631128, 0.56631128,
        0.56523145, 0.56523145, 0.56523145, 0.56523145, 0.56523145,
        0.56458828, 0.56458828, 0.56458828, 0.56458828, 0.56458828]),
 'std_test_score': array([0.0695323 , 0.0695323 , 0.0695323 , 0.0695323 ,
0.0695323 ,
        0.07670739, 0.07670739, 0.07670739, 0.07670739, 0.07670739,
        0.07721956, 0.07721956, 0.07721956, 0.07721956, 0.07721956,
        0.07308592, 0.07308592, 0.07308592, 0.07308592, 0.07308592,
        0.07671734, 0.07671734, 0.07671734, 0.07671734, 0.07671734]),
 'rank_test_score': array([ 1,  1,  1,  1,  1,  6,  6,  6,  6,  6, 11, 11, 11,
11, 11, 16, 16,
        16, 16, 16, 21, 21, 21, 21, 21], dtype=int32),
 'split0_train_score': array([0.63541439, 0.63541439, 0.63541439, 0.63541439,
0.63541439,
        0.63413401, 0.63413401, 0.63413401, 0.63413401, 0.63413401,
        0.63542945, 0.63542945, 0.63542945, 0.63542945, 0.63542945,
        0.63428559, 0.63428559, 0.63428559, 0.63428559, 0.63428559,
        0.63630501, 0.63630501, 0.63630501, 0.63630501, 0.63630501]),
 'split1_train_score': array([0.65606084, 0.65606084, 0.65606084, 0.65606084,
0.65606084,
        0.65556378, 0.65556378, 0.65556378, 0.65556378, 0.65556378,
        0.65620393, 0.65620393, 0.65620393, 0.65620393, 0.65620393,
        0.65586691, 0.65586691, 0.65586691, 0.65586691, 0.65586691,
        0.65609473, 0.65609473, 0.65609473, 0.65609473, 0.65609473]),
 'split2_train_score': array([0.68089124, 0.68089124, 0.68089124, 0.68089124,
0.68089124,
        0.68099009, 0.68099009, 0.68099009, 0.68099009, 0.68099009,
        0.68717039, 0.68717039, 0.68717039, 0.68717039, 0.68717039,
        0.68436313, 0.68436313, 0.68436313, 0.68436313, 0.68436313,
        0.68686349, 0.68686349, 0.68686349, 0.68686349, 0.68686349]),
 'split3_train_score': array([0.64233508, 0.64233508, 0.64233508, 0.64233508,
0.64233508,
        0.64254168, 0.64254168, 0.64254168, 0.64254168, 0.64254168,
        0.64174586, 0.64174586, 0.64174586, 0.64174586, 0.64174586,
        0.63948993, 0.63948993, 0.63948993, 0.63948993, 0.63948993,
        0.6424092 , 0.6424092 , 0.6424092 , 0.6424092 , 0.6424092 ]),
 'split4_train_score': array([0.6184343 , 0.6184343 , 0.6184343 , 0.6184343 ,
0.6184343 ,
        0.61847785, 0.61847785, 0.61847785, 0.61847785, 0.61847785,
        0.61798775, 0.61798775, 0.61798775, 0.61798775, 0.61798775,
        0.61879655, 0.61879655, 0.61879655, 0.61879655, 0.61879655,
        0.61812857, 0.61812857, 0.61812857, 0.61812857, 0.61812857]),
```

```
 'mean_train_score': array([0.64662717, 0.64662717, 0.64662717, 0.64662717,
0.64662717,
        0.64634148, 0.64634148, 0.64634148, 0.64634148, 0.64634148,
        0.64770748, 0.64770748, 0.64770748, 0.64770748, 0.64770748,
        0.64656042, 0.64656042, 0.64656042, 0.64656042, 0.64656042,
        0.6479602 , 0.6479602 , 0.6479602 , 0.6479602 , 0.6479602 ]),
 'std_train_score': array([0.02098574, 0.02098574, 0.02098574, 0.02098574,
0.02098574,
        0.02109701, 0.02109701, 0.02109701, 0.02109701, 0.02109701,
        0.02323387, 0.02323387, 0.02323387, 0.02323387, 0.02323387,
        0.02230315, 0.02230315, 0.02230315, 0.02230315, 0.02230315,
        0.02296179, 0.02296179, 0.02296179, 0.02296179, 0.02296179])}
```

[26]:
```
lg1.best_estimator_
```

[26]:
```
LogisticRegression(C=0.5, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='multinomial', n_jobs=None, penalty='l2',
                   random_state=0, solver='lbfgs', tol=1e-06, verbose=0,
                   warm_start=False)
```

[27]:
```
y_pred2 = lg1.best_estimator_.predict(read_Xtest2)
y_pred2
```

[27]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[28]:
```
prob2 = lg1.best_estimator_.predict_proba(read_Xtest2)
prob2
```

[28]:
```
array([[0.98997186, 0.01002814],
       [0.99208034, 0.00791966],
       [0.99414131, 0.00585869],
       …,
       [0.98861509, 0.01138491],
       [0.99548535, 0.00451465],
       [0.99319804, 0.00680196]])
```

[29]:
```
lg_matrix = metrics.confusion_matrix(read_ytest2, y_pred2)
lg_matrix
```

[29]:
```
array([[14599,     0],
       [   50,     0]])
```

[30]:
```
target_names1 = ['Not in 30 days', 'Readmitted within 30 days']
print("", classification_report(read_ytest2, y_pred2,␣
 ↪target_names=target_names1))
```

```
                         precision    recall  f1-score   support

           Not in 30 days       1.00      1.00      1.00     14599
Readmitted within 30 days       0.00      0.00      0.00        50

                 accuracy                           1.00     14649
                macro avg       0.50      0.50      0.50     14649
             weighted avg       0.99      1.00      0.99     14649
```

```
/usr/local/lib/python3.7/site-packages/sklearn/metrics/classification.py:1437:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples.
  'precision', 'predicted', average, warn_for)
```

[31]:
```
lg_probs = lg1.best_estimator_.predict_proba(read_Xtest2)[:,1]
print(roc_auc_score(read_ytest2, lg_probs))
```

```
0.47788067675868207
```

[32]:
```
import scipy.stats
def mean_confidence_interval(data, confidence=0.95):
    a = 1.0 * np.array(data)
    n = len(a)
    m, se = np.mean(a), scipy.stats.sem(a)
    h = se * scipy.stats.t.ppf((1 + confidence) / 2., n-1)
    return m, m-h, m+h
mean_confidence_interval(read_ytest2, confidence=0.95)
```

[32]:
```
(0.003413202266366305, array([0.00246863]), array([0.00435777]))
```

[33]:
```
import numpy as np, scipy.stats as st
st.t.interval(0.95, len(read_ytest2)-1, loc=np.mean(read_ytest2), scale=st.
 ↪sem(read_ytest2))
```

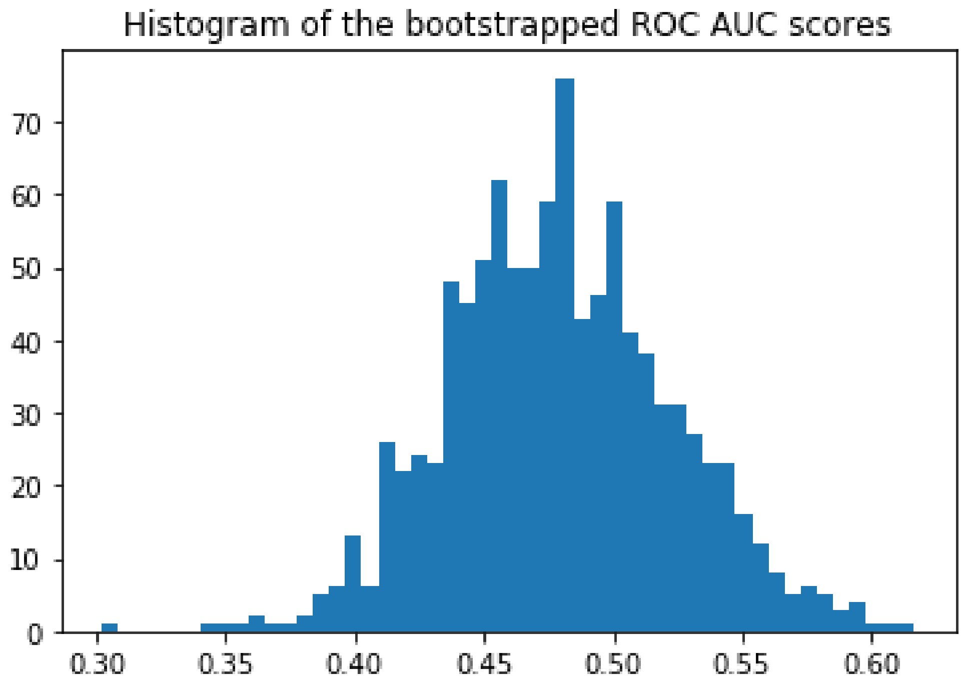
[33]:
```
(array([0.00246863]), array([0.00435777]))
```

[34]:
```
read_ytest2=read_ytest2.values
```

[35]:
```
#Boostrapping calculated 95% CI
y_pred = lg_probs
y_true = read_ytest2

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []
```

```python
rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.4779
```




```
Confidence interval for the score: [0.4117 - 0.5528]
```

[36]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
read_ytest2=read_ytest2.reshape((14649,))
y_pred = lg_probs
y_true = read_ytest2

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.47788067675868207
AUC COV: 0.0019309876555635688
95% AUC CI: [0.39175397 0.56400738]
```

[ ]:

5 Artificial Neural Network

5.1 SMOTE First

[37]:
```
# !pip install imblearn
```

[38]:
```
from imblearn.over_sampling import SMOTE
```

```
Using TensorFlow backend.
```

[39]:
```
smt = SMOTE()
X_train = read_Xtrain2
X_test = read_Xtest2
y_train = read_ytrain2
y_test = read_ytest2
X_train, y_train = smt.fit_sample(X_train, y_train)
```

```
/usr/local/lib/python3.7/site-packages/sklearn/utils/validation.py:724:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

5.1.1 To test for the SMOTE

[40]:
```
read_ytrain2.shape
```

[40]:
```
(21494, 1)
```

[41]:
```
read_ytrain2[read_ytrain2['outcome']==0].shape
```

[41]:
```
(21416, 1)
```

[42]:
```
read_ytrain2[read_ytrain2['outcome']==1].shape
```

[42]:
```
(78, 1)
```

[43]:
```
X_train.shape
```

[43]: (42832, 19)

[44]:
```
y_train = pd.DataFrame(y_train)
y_train.columns = ['outcome']
y_train[y_train['outcome']==0].shape
```

[44]: (21416, 1)

[45]:
```
y_train[y_train['outcome']==1].shape
```

[45]: (21416, 1)

[46]:
```
X_train.shape
```

[46]: (42832, 19)

5.2 ANN from here

[47]:
```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
# Fit only to the training data
scaler = scaler.fit(X_train)
X_trains = scaler.transform(X_train)
X_tests = scaler.transform(X_test)
# y_train
# y_test
```

[48]:
```
from sklearn.neural_network import MLPClassifier
seed(1)
# skf = StratifiedKFold(n_splits=5)
params3 = {'alpha' : [0.0001,0.01],
           'power_t': [0.5,0.75],
           'max_iter': [200,250]}
mlp1 = MLPClassifier(solver='lbfgs', random_state=1)
mlp1 = GridSearchCV(mlp1, cv=5, param_grid=params3, scoring = 'roc_auc',refit =␣
 ↪True,
                    n_jobs=-1, verbose = 5, return_train_score=True)

mlp1.fit(X_trains, y_train)
mlp1.cv_results_
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 out of  40 | elapsed:  1.0min remaining:   32.9s
[Parallel(n_jobs=-1)]: Done  35 out of  40 | elapsed:  1.1min remaining:    9.1s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:  1.2min finished
```

```
/usr/local/lib/python3.7/site-
packages/sklearn/neural_network/multilayer_perceptron.py:921:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

[48]:

```
{'mean_fit_time': array([20.03796577, 20.12409124, 19.96209979, 19.84366527,
19.97163229,
        19.32875395, 19.26228871, 10.5093688 ]),
 'std_fit_time': array([0.53757763, 0.55288829, 0.40897657, 0.58283489,
0.96777147,
        1.03007631, 0.75743051, 4.11790708]),
 'mean_score_time': array([0.07484741, 0.07677236, 0.08022375, 0.07389636,
0.07059608,
        0.06423168, 0.05200181, 0.02085433]),
 'std_score_time': array([0.01192374, 0.00756909, 0.01119031, 0.0113187 ,
0.00645661,
        0.01178538, 0.01444176, 0.00624069]),
 'param_alpha': masked_array(data=[0.0001, 0.0001, 0.0001, 0.0001, 0.01, 0.01,
0.01, 0.01],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_max_iter': masked_array(data=[200, 200, 250, 250, 200, 200, 250, 250],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_power_t': masked_array(data=[0.5, 0.75, 0.5, 0.75, 0.5, 0.75, 0.5,
0.75],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'alpha': 0.0001, 'max_iter': 200, 'power_t': 0.5},
  {'alpha': 0.0001, 'max_iter': 200, 'power_t': 0.75},
  {'alpha': 0.0001, 'max_iter': 250, 'power_t': 0.5},
  {'alpha': 0.0001, 'max_iter': 250, 'power_t': 0.75},
  {'alpha': 0.01, 'max_iter': 200, 'power_t': 0.5},
  {'alpha': 0.01, 'max_iter': 200, 'power_t': 0.75},
  {'alpha': 0.01, 'max_iter': 250, 'power_t': 0.5},
  {'alpha': 0.01, 'max_iter': 250, 'power_t': 0.75}],
 'split0_test_score': array([0.99908634, 0.99908634, 0.99908634, 0.99908634,
0.99889635,
        0.99889635, 0.99889635, 0.99889635]),
 'split1_test_score': array([0.99903037, 0.99903037, 0.99903037, 0.99903037,
0.99922885,
        0.99922885, 0.99922885, 0.99922885]),
```

```
 'split2_test_score': array([0.9992324 , 0.9992324 , 0.9992324 , 0.9992324 ,
0.99948152,
        0.99948152, 0.99948152, 0.99948152]),
 'split3_test_score': array([0.99929689, 0.99929689, 0.99929689, 0.99929689,
0.9993037 ,
        0.9993037 , 0.9993037 , 0.9993037 ]),
 'split4_test_score': array([0.99919571, 0.99919571, 0.99919571, 0.99919571,
0.99932289,
        0.99932289, 0.99932289, 0.99932289]),
 'mean_test_score': array([0.99916834, 0.99916834, 0.99916834, 0.99916834,
0.99924665,
        0.99924665, 0.99924665, 0.99924665]),
 'std_test_score': array([9.70923890e-05, 9.70923890e-05, 9.70923890e-05,
9.70923890e-05,
        1.93536636e-04, 1.93536636e-04, 1.93536636e-04, 1.93536636e-04]),
 'rank_test_score': array([5, 5, 5, 5, 1, 1, 1, 1], dtype=int32),
 'split0_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
 'split1_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
 'split2_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
 'split3_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
 'split4_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
 'mean_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
 'std_train_score': array([0., 0., 0., 0., 0., 0., 0., 0.])}
```

[49]:
```
mlp1.best_estimator_
```

[49]:
```
MLPClassifier(activation='relu', alpha=0.01, batch_size='auto', beta_1=0.9,
              beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes=(100,), learning_rate='constant',
              learning_rate_init=0.001, max_iter=200, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
              random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
```

[50]:
```
# from sklearn.neural_network import MLPClassifier
# mlp = MLPClassifier(solver='lbfgs', random_state=1)
# mlp = mlp.fit(X_trains, y_train)
# ann_pred1 = mlp.predict(X_tests)
# ann_pred1
```

[51]:
```
# mlp
```

[52]:
```
ann_pred1 = mlp1.best_estimator_.predict(X_tests)
ann_pred1
```

[52]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[53]:
```
prob3 = mlp1.best_estimator_.predict_proba(X_tests)
prob3
```

[53]:
```
array([[1.00000000e+00, 6.60463288e-12],
       [1.00000000e+00, 9.96650943e-30],
       [1.00000000e+00, 4.58060948e-41],
       …,
       [1.00000000e+00, 6.68074818e-55],
       [1.00000000e+00, 1.34052674e-13],
       [1.00000000e+00, 4.50683857e-49]])
```

[54]:
```
ann_matrix = metrics.confusion_matrix(y_test, ann_pred1)
ann_matrix
```

[54]:
```
array([[14496,   103],
       [   50,     0]])
```

[55]:
```
target_names1 = ['Not in 30 days', 'Readmitted within 30 days']
print("", classification_report(y_test, ann_pred1,
                                target_names=target_names1))
```

```
                            precision    recall  f1-score   support

            Not in 30 days       1.00      0.99      0.99     14599
Readmitted within 30 days       0.00      0.00      0.00        50

                  accuracy                           0.99     14649
                 macro avg       0.50      0.50      0.50     14649
              weighted avg       0.99      0.99      0.99     14649
```

[56]:
```
ann_probs = mlp1.best_estimator_.predict_proba(X_tests)[:,1]
print(roc_auc_score(y_test, ann_probs))
```

```
0.5075183231728201
```

[57]:
```
mean_confidence_interval(y_test, confidence=0.95)
```

[57]:
```
(0.003413202266366305, 0.0024686339150578173, 0.004357770617674793)
```

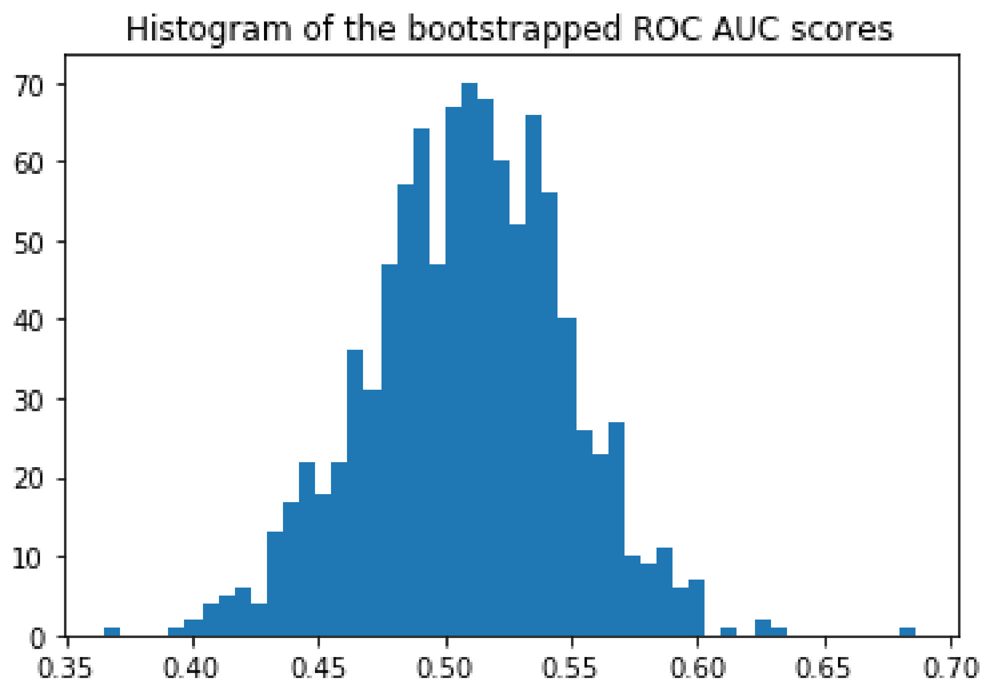
[58]:
```
st.t.interval(0.95, len(y_test)-1,
              loc=np.mean(y_test), scale=st.sem(y_test))
```

[58]:
```
(0.0024686339150578173, 0.004357770617674793)
```

[59]:
```
#Boostrapping calculated 95% CI
y_pred = ann_probs
```

```
y_true = y_test

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.5075
```




```
Confidence interval for the score: [0.4412 - 0.5704]
```

[60]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
y_pred = ann_probs
y_true = y_test

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.5075183231728201
AUC COV: 0.001641132475367912
95% AUC CI: [0.42811843 0.58691822]
```

[ ]:

6 Random Forest

[62]:
```
# read_Xtrain2
# read_Xtest2
# read_ytrain2
# read_ytest2
```

[63]:
```
from sklearn.ensemble import RandomForestClassifier
seed(42)

params4 = {'n_estimators' : [10,100,150],
           'min_samples_leaf': [1,2,3]}
rf1 = RandomForestClassifier(random_state=42)
rf1 = GridSearchCV(rf1, cv=5, param_grid=params4, scoring = 'roc_auc',refit =␣
 ↪True,
                   n_jobs=-1, verbose = 5, return_train_score=True)

rf1.fit(read_Xtrain2, read_ytrain2)
rf1.cv_results_
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 out of  45 | elapsed:   10.6s remaining:    4.3s
[Parallel(n_jobs=-1)]: Done  42 out of  45 | elapsed:   13.0s remaining:    0.9s
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed:   14.3s finished
/usr/local/lib/python3.7/site-packages/sklearn/model_selection/_search.py:715:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
  self.best_estimator_.fit(X, y, **fit_params)
```

[63]:
```
{'mean_fit_time': array([0.28285446, 2.79317765, 4.17807322, 0.28368745,
2.84362769,
        4.27447343, 0.29339499, 2.88624263, 3.28689208]),
 'std_fit_time': array([0.00912064, 0.06771047, 0.10350626, 0.00309684,
0.12585007,
        0.11082721, 0.01176973, 0.03636681, 0.31933991]),
 'mean_score_time': array([0.0130074 , 0.06990938, 0.10596676, 0.01274948,
0.07116423,
```

```
       0.09457936, 0.01342211, 0.06731086, 0.0575665 ]),
 'std_score_time': array([0.00027365, 0.00119611, 0.00156421, 0.00044756,
0.00391362,
       0.01012576, 0.00032316, 0.00848722, 0.00418184]),
 'param_min_samples_leaf': masked_array(data=[1, 1, 1, 2, 2, 2, 3, 3, 3],
             mask=[False, False, False, False, False, False, False, False,
                   False],
       fill_value='?',
            dtype=object),
 'param_n_estimators': masked_array(data=[10, 100, 150, 10, 100, 150, 10, 100,
150],
             mask=[False, False, False, False, False, False, False, False,
                   False],
       fill_value='?',
            dtype=object),
 'params': [{'min_samples_leaf': 1, 'n_estimators': 10},
  {'min_samples_leaf': 1, 'n_estimators': 100},
  {'min_samples_leaf': 1, 'n_estimators': 150},
  {'min_samples_leaf': 2, 'n_estimators': 10},
  {'min_samples_leaf': 2, 'n_estimators': 100},
  {'min_samples_leaf': 2, 'n_estimators': 150},
  {'min_samples_leaf': 3, 'n_estimators': 10},
  {'min_samples_leaf': 3, 'n_estimators': 100},
  {'min_samples_leaf': 3, 'n_estimators': 150}],
 'split0_test_score': array([0.47840803, 0.41719188, 0.41306314, 0.52016953,
0.45726103,
       0.44823033, 0.50838877, 0.53085609, 0.54745857]),
 'split1_test_score': array([0.47863647, 0.41943439, 0.4376897 , 0.5237859 ,
0.4980373 ,
       0.44794099, 0.5428438 , 0.50035752, 0.48863968]),
 'split2_test_score': array([0.50912036, 0.44322029, 0.45397502, 0.49720552,
0.51347624,
       0.47672484, 0.48102965, 0.48605679, 0.47062515]),
 'split3_test_score': array([0.47805277, 0.47760915, 0.56379485, 0.49603082,
0.6091369 ,
       0.6142657 , 0.45155265, 0.57443381, 0.59211612]),
 'split4_test_score': array([0.51095027, 0.49101876, 0.48364075, 0.46299323,
0.63738034,
       0.62428983, 0.4491011 , 0.49220173, 0.55887618]),
 'mean_test_score': array([0.49103267, 0.44969016, 0.47042506, 0.50003985,
0.54304691,
       0.52227787, 0.48658758, 0.5167803 , 0.53153979]),
 'std_test_score': array([0.01552651, 0.0300027 , 0.05200864, 0.02175094,
0.06859433,
       0.0799383 , 0.03550611, 0.03267777, 0.04521216]),
 'rank_test_score': array([6, 9, 8, 5, 1, 3, 7, 4, 2], dtype=int32),
 'split0_train_score': array([0.99999812, 1.        , 1.        , 0.99994587, 1.
```

```
,
        1.        , 0.9999421 , 1.        , 1.        ]),
 'split1_train_score': array([1.        , 1.        , 1.        , 0.99998917, 1.
,
        1.        , 0.99994163, 1.        , 1.        ]),
 'split2_train_score': array([0.99999153, 1.        , 1.        , 0.99990256, 1.
,
        1.        , 0.99969828, 1.        , 1.        ]),
 'split3_train_score': array([0.99999027, 1.        , 1.        , 0.99991708, 1.
,
        1.        , 0.99905131, 1.        , 1.        ]),
 'split4_train_score': array([0.99998147, 1.        , 1.        , 0.9995289 , 1.
,
        1.        , 0.99952148, 1.        , 1.        ]),
 'mean_train_score': array([0.99999228, 1.        , 1.        , 0.99985672, 1.
,
        1.        , 0.99963096, 1.        , 1.        ]),
 'std_train_score': array([6.55905362e-06, 0.00000000e+00, 0.00000000e+00,
1.66556373e-04,
        0.00000000e+00, 4.96506831e-17, 3.30406282e-04, 8.59975057e-17,
        1.11022302e-16])}
```

[64]:
```
rf1.best_estimator_
```

[64]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=2, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

[65]:
```
rf_pred1 = rf1.best_estimator_.predict(read_Xtest2)
rf_pred1
```

[65]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[66]:
```
prob4 = rf1.best_estimator_.predict_proba(read_Xtest2)
prob4
```

[66]:
```
array([[0.99666667, 0.00333333],
       [0.975     , 0.025     ],
       [0.94866667, 0.05133333],
       …,
       [1.        , 0.        ],
       [0.99333333, 0.00666667],
       [1.        , 0.        ]])
```

[67]:
```python
rf_matrix1 = metrics.confusion_matrix(read_ytest2, rf_pred1)
rf_matrix1
```

[67]:
```
array([[14599,     0],
       [   50,     0]])
```

[68]:
```python
target_names1 = ['Not in 30 days', 'Readmitted within 30 days']
print("", classification_report(read_ytest2, rf_pred1,
                                target_names=target_names1))
```

```
                           precision    recall  f1-score   support

           Not in 30 days       1.00      1.00      1.00     14599
Readmitted within 30 days       0.00      0.00      0.00        50

                 accuracy                           1.00     14649
                macro avg       0.50      0.50      0.50     14649
             weighted avg       0.99      1.00      0.99     14649
```

```
/usr/local/lib/python3.7/site-packages/sklearn/metrics/classification.py:1437:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples.
  'precision', 'predicted', average, warn_for)
```

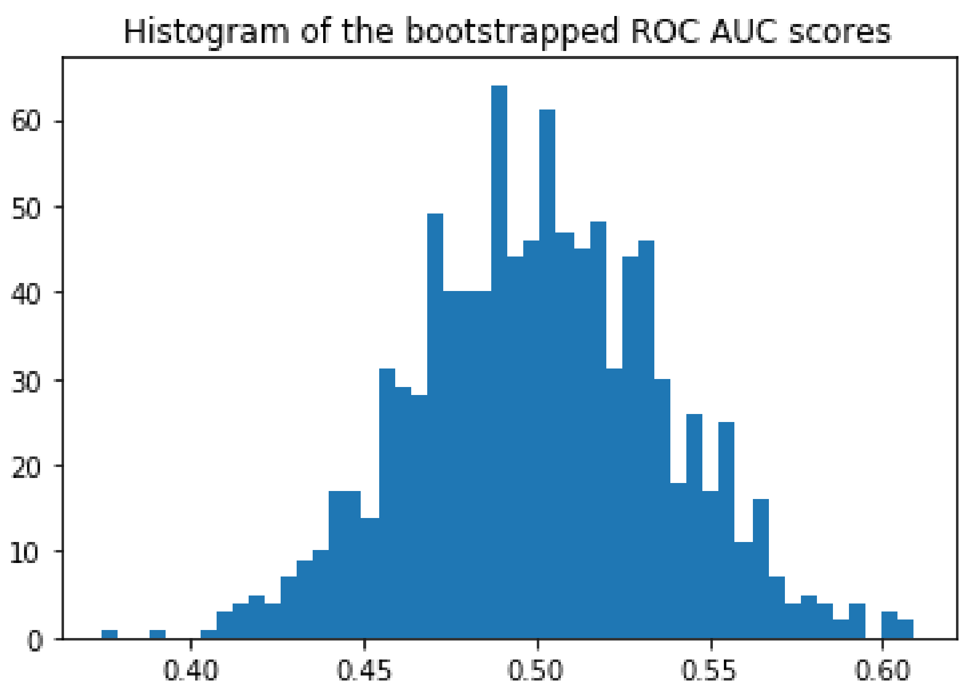
[69]:
```python
rf_probs = rf1.best_estimator_.predict_proba(read_Xtest2)[:,1]
print(roc_auc_score(read_ytest2, rf_probs))
```

```
0.5017823138571136
```

[70]:
```python
#Boostrapping calculated 95% CI
y_pred = rf_probs
y_true = read_ytest2

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue
```

```
    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.5018
```




```
Confidence interval for the score: [0.4423 - 0.5618]
```

[71]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
y_pred = rf_probs
y_true = read_ytest2

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.5017823138571135
AUC COV: 0.0013611946962794295
95% AUC CI: [0.42947067 0.57409395]
```

[ ]:
```
```

7 Gradient Boosting Machines

[72]:
```
read_Xtrain2.head()
```

[72]:
```
    PatientGender  PatientRace  PatientEncounterAge  \
0               1            1            36.456455
1               1            1            21.408437
3               1            1            42.671151
9               1            3            27.803538
12              1            1            66.119312

    CBC..ABSOLUTE.LYMPHOCYTES  CBC..ABSOLUTE.NEUTROPHILS  CBC..BASOPHILS  \
0                        16.1                       77.9             0.2
1                        32.8                       76.4             0.0
3                        23.4                       66.2             0.1
9                        29.4                       64.5             0.0
```

```
12                   22.4                           77.5            0.1

    CBC..EOSINOPHILS  CBC..HEMATOCRIT  CBC..HEMOGLOBIN  CBC..PLATELET.COUNT  \
0                0.4             49.6             16.2                339.2
1                0.1             35.9             14.8                386.0
3                0.4             40.4             18.7                408.7
9                0.3             42.9             13.4                271.7
12               0.2             39.3             13.3                135.0

    CBC..RED.BLOOD.CELL.COUNT  CBC..WHITE.BLOOD.CELL.COUNT  \
0                         5.8                          6.4
1                         4.8                         10.7
3                         3.2                          8.5
9                         5.9                          6.4
12                        5.3                          3.0

    METABOLIC..ALBUMIN  METABOLIC..BILI.TOTAL  METABOLIC..BUN  \
0                  2.7                    0.9            15.6
1                  5.8                    0.8            12.6
3                  3.7                    0.3            15.1
9                  3.2                    0.0            25.9
12                 5.7                    1.1            25.9

    METABOLIC..CALCIUM  METABOLIC..CREATININE  METABOLIC..POTASSIUM  \
0                  8.1                    1.2                   4.7
1                 11.1                    0.9                   4.1
3                  7.9                    0.7                   5.3
9                 11.8                    0.7                   3.3
12                11.1                    1.2                   4.7

    METABOLIC..SODIUM
0               140.9
1               135.5
3               150.0
9               145.2
12              136.0
```

[73]:

```
from sklearn.ensemble import GradientBoostingClassifier
seed(10)

params5 = {'learning_rate' : [0.05,0.1,0.2],
           'n_estimators': [100,150],
           'min_samples_split': [2,3,4]}
gbm1 = GradientBoostingClassifier(random_state=10)
gbm1 = GridSearchCV(gbm1, cv=5, param_grid=params5, scoring = 'roc_auc',refit = True,
                    n_jobs=-1, verbose = 5, return_train_score=True)
```

```
gbm1.fit(read_Xtrain2, read_ytrain2)
gbm1.cv_results_
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:   21.9s
[Parallel(n_jobs=-1)]: Done  86 out of  90 | elapsed:   34.8s remaining:    1.6s
[Parallel(n_jobs=-1)]: Done  90 out of  90 | elapsed:   36.0s finished
/usr/local/lib/python3.7/site-
packages/sklearn/ensemble/gradient_boosting.py:1450: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

[73]:
```
{'mean_fit_time': array([3.14430537, 4.66901126, 3.22242279, 5.04610419,
3.490838  ,
        5.6182436 , 3.71450071, 5.5807796 , 3.64367723, 5.26027641,
        3.47499132, 5.1421937 , 3.42881041, 5.19725356, 3.45074306,
        5.11695271, 3.34255419, 3.27638688]),
 'std_fit_time': array([0.04620515, 0.05551474, 0.09116727, 0.06166162,
0.06277106,
        0.13388481, 0.05391693, 0.09711928, 0.0848055 , 0.06619662,
        0.08144634, 0.07152544, 0.07926821, 0.09208795, 0.01771899,
        0.11685436, 0.04269919, 0.6721486 ]),
 'mean_score_time': array([0.01672258, 0.02444882, 0.01834774, 0.02433524,
0.01909771,
        0.02615447, 0.01880374, 0.02562137, 0.01856661, 0.02241302,
        0.01907263, 0.02244263, 0.0200892 , 0.02315779, 0.01781116,
        0.01922665, 0.01633358, 0.01152754]),
 'std_score_time': array([0.00034095, 0.00249744, 0.00140721, 0.00146483,
0.00143651,
        0.00221613, 0.00085346, 0.00197047, 0.00059951, 0.00050705,
        0.00165574, 0.00061754, 0.00377675, 0.00183234, 0.00089573,
        0.0034698 , 0.00082567, 0.00089842]),
 'param_learning_rate': masked_array(data=[0.05, 0.05, 0.05, 0.05, 0.05, 0.05,
0.1, 0.1, 0.1, 0.1,
                    0.1, 0.1, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_min_samples_split': masked_array(data=[2, 2, 3, 3, 4, 4, 2, 2, 3, 3, 4,
4, 2, 2, 3, 3, 4, 4],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
```

```
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[100, 150, 100, 150, 100, 150, 100,
150, 100, 150, 100,
                    150, 100, 150, 100, 150, 100, 150],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'learning_rate': 0.05,
   'min_samples_split': 2,
   'n_estimators': 100},
  {'learning_rate': 0.05, 'min_samples_split': 2, 'n_estimators': 150},
  {'learning_rate': 0.05, 'min_samples_split': 3, 'n_estimators': 100},
  {'learning_rate': 0.05, 'min_samples_split': 3, 'n_estimators': 150},
  {'learning_rate': 0.05, 'min_samples_split': 4, 'n_estimators': 100},
  {'learning_rate': 0.05, 'min_samples_split': 4, 'n_estimators': 150},
  {'learning_rate': 0.1, 'min_samples_split': 2, 'n_estimators': 100},
  {'learning_rate': 0.1, 'min_samples_split': 2, 'n_estimators': 150},
  {'learning_rate': 0.1, 'min_samples_split': 3, 'n_estimators': 100},
  {'learning_rate': 0.1, 'min_samples_split': 3, 'n_estimators': 150},
  {'learning_rate': 0.1, 'min_samples_split': 4, 'n_estimators': 100},
  {'learning_rate': 0.1, 'min_samples_split': 4, 'n_estimators': 150},
  {'learning_rate': 0.2, 'min_samples_split': 2, 'n_estimators': 100},
  {'learning_rate': 0.2, 'min_samples_split': 2, 'n_estimators': 150},
  {'learning_rate': 0.2, 'min_samples_split': 3, 'n_estimators': 100},
  {'learning_rate': 0.2, 'min_samples_split': 3, 'n_estimators': 150},
  {'learning_rate': 0.2, 'min_samples_split': 4, 'n_estimators': 100},
  {'learning_rate': 0.2, 'min_samples_split': 4, 'n_estimators': 150}],
 'split0_test_score': array([0.5146767 , 0.55931956, 0.5146767 , 0.55931956,
0.51768937,
        0.5383841 , 0.54089344, 0.59010271, 0.54086426, 0.59935224,
        0.54515348, 0.60031513, 0.56522817, 0.58972339, 0.565199  ,
        0.58969421, 0.61188434, 0.58873133]),
 'split1_test_score': array([0.50161978, 0.50515118, 0.50159059, 0.50767569,
0.50159059,
        0.50767569, 0.58106176, 0.53318352, 0.58110553, 0.53324189,
        0.56912503, 0.54497432, 0.4893912 , 0.50062748, 0.49655615,
        0.49384193, 0.49649778, 0.49375438]),
 'split2_test_score': array([0.53972099, 0.53598529, 0.53967721, 0.53410285,
0.5396918 ,
        0.53411744, 0.52218071, 0.53815958, 0.48448809, 0.53350455,
        0.49511149, 0.52911219, 0.65374737, 0.66561114, 0.50809888,
        0.54040684, 0.62594852, 0.61103491]),
 'split3_test_score': array([0.64050121, 0.64492178, 0.65421434, 0.65343607,
```

```
0.68212312,
       0.65421434, 0.55744416, 0.54857187, 0.55744416, 0.54857187,
       0.55744416, 0.54858744, 0.54063351, 0.52688925, 0.51306716,
       0.4459958 , 0.51311386, 0.44602693]),
 'split4_test_score': array([0.63769165, 0.64884427, 0.62306794, 0.61365087,
0.61832049,
       0.62949646, 0.67742237, 0.70422601, 0.63689003, 0.65457234,
       0.63447739, 0.65757647, 0.56851117, 0.55870496, 0.53680442,
       0.55258775, 0.61397774, 0.5871274 ]),
 'mean_test_score': array([0.56683291, 0.57883718, 0.56663624, 0.57363077,
0.57187326,
       0.57276958, 0.57579499, 0.58284502, 0.56015407, 0.57384719,
       0.56025828, 0.57611172, 0.5635032 , 0.56831461, 0.52394695,
       0.52451069, 0.5722871 , 0.54533968]),
 'std_test_score': array([0.06025795, 0.05816139, 0.06084561, 0.05307859,
0.06814984,
       0.05790253, 0.05437594, 0.06390772, 0.0498627 , 0.047077  ,
       0.04486984, 0.04723838, 0.05326952, 0.05712328, 0.024443  ,
       0.04978958, 0.0555535 , 0.06398073]),
 'rank_test_score': array([11,  2, 12,  6,  9,  7,  4,  1, 15,  5, 14,  3, 13,
10, 18, 17,  8,
       16], dtype=int32),
 'split0_train_score': array([0.98599866, 0.99703912, 0.98599866, 0.99703912,
0.98453846,
       0.99355573, 0.99689319, 0.99995669, 0.99689131, 0.99986161,
       0.99945866, 0.99999812, 0.99995952, 1.        , 0.99995952,
       1.        , 0.9992619 , 0.99942383]),
 'split1_train_score': array([0.98006488, 0.99173261, 0.98006394, 0.99297903,
0.98006394,
       0.99297903, 0.99602352, 0.99938056, 0.99602352, 0.99938056,
       0.99850505, 0.99998305, 0.98291356, 0.9833532 , 0.98303642,
       0.98359043, 0.98303642, 0.98359043]),
 'split2_train_score': array([0.989413  , 0.99719368, 0.98941206, 0.99719556,
0.98941017,
       0.99717862, 0.98251817, 0.98388321, 0.99192654, 0.99837797,
       0.99724828, 0.99972135, 0.98396417, 0.99994163, 0.98342663,
       1.        , 0.98394157, 0.98371187]),
 'split3_train_score': array([0.9886972 , 0.99782468, 0.98879309, 0.99830829,
0.98680352,
       0.99815635, 0.99451259, 0.99997313, 0.99451259, 0.99997313,
       0.99451259, 0.99997313, 0.99998703, 1.        , 0.99942096,
       0.99976653, 0.99942096, 0.99976653]),
 'split4_train_score': array([0.98967832, 0.99781078, 0.99146732, 0.99868443,
0.99273008,
       0.99828513, 0.99858159, 1.        , 0.99850748, 0.99996757,
       0.99770701, 0.999874  , 0.99896607, 1.        , 0.99988975,
       1.        , 0.99996016, 1.        ]),
```

```
 'mean_train_score': array([0.98677041, 0.99632017, 0.98714701, 0.99684129,
0.98670924,
        0.99603097, 0.99370581, 0.99663872, 0.99557229, 0.99951217,
        0.99748632, 0.99990993, 0.99315807, 0.99665897, 0.99314665,
        0.99667139, 0.9931242 , 0.99329853]),
 'std_train_score': array([0.00359754, 0.00231564, 0.00394952, 0.0020313 ,
0.00429946,
        0.00229594, 0.00574663, 0.00638195, 0.00223487, 0.00060757,
        0.00166592, 0.0001039 , 0.00795116, 0.00665292, 0.00809873,
        0.00654111, 0.00787572, 0.00787928])}
```

[74]:
```
gbm1.best_estimator_
```

[74]:
```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.1, loss='deviance', max_depth=3,
                           max_features=None, max_leaf_nodes=None,
                           min_impurity_decrease=0.0, min_impurity_split=None,
                           min_samples_leaf=1, min_samples_split=2,
                           min_weight_fraction_leaf=0.0, n_estimators=150,
                           n_iter_no_change=None, presort='auto',
                           random_state=10, subsample=1.0, tol=0.0001,
                           validation_fraction=0.1, verbose=0,
                           warm_start=False)
```

[75]:
```
gbm_pred1 = gbm1.best_estimator_.predict(read_Xtest2)
gbm_pred1
```

[75]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[76]:
```
prob5 = gbm1.best_estimator_.predict_proba(read_Xtest2)
prob5
```

[76]:
```
array([[9.99375485e-01, 6.24515243e-04],
       [9.93204431e-01, 6.79556866e-03],
       [9.97519579e-01, 2.48042053e-03],
       …,
       [9.99899119e-01, 1.00881367e-04],
       [9.98525010e-01, 1.47499011e-03],
       [9.99034825e-01, 9.65174759e-04]])
```

[77]:
```
gbm_matrix1 = metrics.confusion_matrix(read_ytest2, gbm_pred1)
gbm_matrix1
```

[77]:
```
array([[14560,    39],
       [   50,     0]])
```

[78]:
```
target_names1 = ['Not in 30 days', 'Readmitted within 30 days']
print("", classification_report(read_ytest2, gbm_pred1,
                                target_names=target_names1))
```

```
                            precision    recall  f1-score   support

            Not in 30 days       1.00      1.00      1.00     14599
Readmitted within 30 days       0.00      0.00      0.00        50

                  accuracy                           0.99     14649
                 macro avg       0.50      0.50      0.50     14649
              weighted avg       0.99      0.99      0.99     14649
```

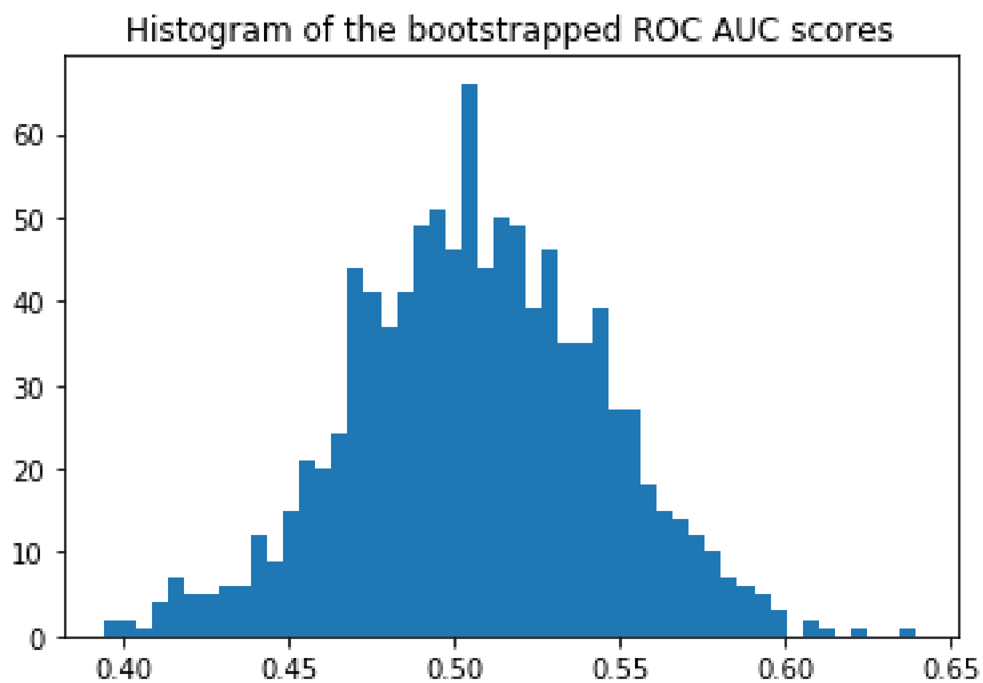
[79]:
```
gbm_probs = gbm1.best_estimator_.predict_proba(read_Xtest2)[:,1]
print(roc_auc_score(read_ytest2, gbm_probs))
```

```
0.5060826083978355
```

[80]:
```
#Boostrapping calculated 95% CI
y_pred = gbm_probs
y_true = read_ytest2

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()
```

```
sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.5061
```




```
Confidence interval for the score: [0.4454 - 0.5693]
```

[81]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
y_pred = gbm_probs
y_true = read_ytest2

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
```

```
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.5060826083978355
AUC COV: 0.001513806136731901
95% AUC CI: [0.42982499 0.58234022]
```

[ ]:

[ ]:

---

8 Section 2: Gusto Study

8.0.1 Using the training datasets, create the following models:

1. GLM model : This model utilizes all features to predict 30-day mortality in a logistic regression framework.
2. Ridge Regression model : This model utilizes all features to predict 30-day mortality in a logistic regression framework with regularization. Utilize a 5 fold cross validation to build the parameters for your model.

9 Gusto

[82]:
```
gusto = pd.read_csv("gusto_data.csv")
gusto['GROUP'] = gusto['GROUP'].replace('west',0)
gusto['GROUP'] = gusto['GROUP'].replace('sample2',1)
gusto['GROUP'] = gusto['GROUP'].replace('sample4',2)
gusto['GROUP'] = gusto['GROUP'].replace('sample5',3)
gusto['GROUP'] = gusto['GROUP'].astype('category').cat.codes
```

[83]:
```
gu_train = gusto.loc[(gusto['GROUP'] == 1
                     ) | (gusto['GROUP'] == 2
```

```
                          ) | (gusto['GROUP'] == 3)]
gu_test = gusto[gusto['GROUP']==0]

gu_Xtrain = gu_train.drop("DAY30",axis=1)
gu_ytrain = gu_train[['DAY30']]
gu_Xtest = gu_test.drop("DAY30",axis=1)
gu_ytest = gu_test[['DAY30']]
```

10 GLM Model

[84]:
```
seed(0)

params6 = {'tol' : [1e-6,1e-5,1e-4,1e-3,1e-2],
           'C': [0.5,1.0,1.5,2.0,2.5]}
lg2 = LogisticRegression(random_state=0, solver='warn',multi_class='warn')
lg2 = GridSearchCV(lg2, cv=5, param_grid=params6, scoring = 'roc_auc',refit =␣
 ↪True,
                   n_jobs=-1, verbose = 5, return_train_score=True)

lg2.fit(gu_Xtrain, gu_ytrain)
lg2.cv_results_
```

```
Fitting 5 folds for each of 25 candidates, totalling 125 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:    2.1s
[Parallel(n_jobs=-1)]: Done 102 out of 125 | elapsed:    2.3s remaining:    0.5s
[Parallel(n_jobs=-1)]: Done 125 out of 125 | elapsed:    2.3s finished
/usr/local/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:432:
FutureWarning: Default solver will be changed to 'lbfgs' in 0.22. Specify a
solver to silence this warning.
  FutureWarning)
/usr/local/lib/python3.7/site-packages/sklearn/utils/validation.py:724:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

[84]:
```
{'mean_fit_time': array([0.01776662, 0.01929483, 0.01552296, 0.01274166,
0.01164804,
        0.01790109, 0.01682034, 0.01415443, 0.01191001, 0.00713024,
        0.01401224, 0.01012397, 0.01103315, 0.00855217, 0.00594544,
        0.0128159 , 0.01101751, 0.01181641, 0.01065273, 0.00775452,
        0.01522622, 0.01626968, 0.01379275, 0.01147804, 0.00932965]),
 'std_fit_time': array([0.00063286, 0.00134065, 0.00224109, 0.00091138,
0.00167021,
```

```
        0.00115815, 0.00095936, 0.00092371, 0.00206895, 0.00104914,
        0.00087095, 0.00051273, 0.00050219, 0.00047801, 0.00064942,
        0.00086447, 0.00156735, 0.00034665, 0.00105303, 0.00113632,
        0.00082124, 0.00299444, 0.00062356, 0.00043167, 0.00114164]),
 'mean_score_time': array([0.00463786, 0.00608678, 0.00451694, 0.00449662,
0.00473275,
        0.00529971, 0.00495882, 0.00479684, 0.0034174 , 0.00309405,
        0.00318289, 0.00240445, 0.00314813, 0.00272183, 0.00283775,
        0.00337129, 0.0033565 , 0.00340214, 0.00347781, 0.00373425,
        0.00367169, 0.00401974, 0.00372605, 0.00333395, 0.00382118]),
 'std_score_time': array([0.00051089, 0.00103407, 0.00052884, 0.00026214,
0.00047231,
        0.00072454, 0.00048463, 0.00026814, 0.000701  , 0.00027082,
        0.0007623 , 0.00016443, 0.00035805, 0.00051427, 0.00084311,
        0.00057566, 0.00062162, 0.00056227, 0.00035981, 0.00066052,
        0.000516  , 0.0004374 , 0.00034133, 0.00075667, 0.0003938 ]),
 'param_C': masked_array(data=[0.5, 0.5, 0.5, 0.5, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0,
1.5,
                    1.5, 1.5, 1.5, 1.5, 2.0, 2.0, 2.0, 2.0, 2.0, 2.5, 2.5,
                    2.5, 2.5, 2.5],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False],
        fill_value='?',
             dtype=object),
 'param_tol': masked_array(data=[1e-06, 1e-05, 0.0001, 0.001, 0.01, 1e-06,
1e-05,
                    0.0001, 0.001, 0.01, 1e-06, 1e-05, 0.0001, 0.001, 0.01,
                    1e-06, 1e-05, 0.0001, 0.001, 0.01, 1e-06, 1e-05,
                    0.0001, 0.001, 0.01],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False],
        fill_value='?',
             dtype=object),
 'params': [{'C': 0.5, 'tol': 1e-06},
  {'C': 0.5, 'tol': 1e-05},
  {'C': 0.5, 'tol': 0.0001},
  {'C': 0.5, 'tol': 0.001},
  {'C': 0.5, 'tol': 0.01},
  {'C': 1.0, 'tol': 1e-06},
  {'C': 1.0, 'tol': 1e-05},
  {'C': 1.0, 'tol': 0.0001},
  {'C': 1.0, 'tol': 0.001},
  {'C': 1.0, 'tol': 0.01},
```

```
  {'C': 1.5, 'tol': 1e-06},
  {'C': 1.5, 'tol': 1e-05},
  {'C': 1.5, 'tol': 0.0001},
  {'C': 1.5, 'tol': 0.001},
  {'C': 1.5, 'tol': 0.01},
  {'C': 2.0, 'tol': 1e-06},
  {'C': 2.0, 'tol': 1e-05},
  {'C': 2.0, 'tol': 0.0001},
  {'C': 2.0, 'tol': 0.001},
  {'C': 2.0, 'tol': 0.01},
  {'C': 2.5, 'tol': 1e-06},
  {'C': 2.5, 'tol': 1e-05},
  {'C': 2.5, 'tol': 0.0001},
  {'C': 2.5, 'tol': 0.001},
  {'C': 2.5, 'tol': 0.01}],
 'split0_test_score': array([0.77952899, 0.77952899, 0.77971014, 0.77663043,
0.775     ,
        0.77608696, 0.77608696, 0.77644928, 0.77300725, 0.77536232,
        0.77572464, 0.77572464, 0.77572464, 0.77192029, 0.77518116,
        0.77463768, 0.77463768, 0.77518116, 0.77264493, 0.77518116,
        0.77445652, 0.77445652, 0.77427536, 0.77463768, 0.77518116]),
 'split1_test_score': array([0.78737605, 0.78756674, 0.78737605, 0.78718535,
0.77517162,
        0.78623188, 0.78623188, 0.78527841, 0.78699466, 0.77459954,
        0.7858505 , 0.7858505 , 0.78546911, 0.78470633, 0.77345538,
        0.78546911, 0.78508772, 0.78546911, 0.78318078, 0.77402746,
        0.78508772, 0.78508772, 0.78451564, 0.78413425, 0.77364607]),
 'split2_test_score': array([0.80076555, 0.80038278, 0.80114833, 0.79425837,
0.80344498,
        0.79961722, 0.79961722, 0.79923445, 0.79923445, 0.80842105,
        0.79885167, 0.79885167, 0.7984689 , 0.79980861, 0.80937799,
        0.79827751, 0.79866029, 0.79866029, 0.79885167, 0.80861244,
        0.79732057, 0.79732057, 0.79751196, 0.78985646, 0.80842105]),
 'split3_test_score': array([0.77665072, 0.77684211, 0.77722488, 0.77626794,
0.76956938,
        0.7737799 , 0.7737799 , 0.7737799 , 0.77492823, 0.784689  ,
        0.77301435, 0.77301435, 0.77320574, 0.77358852, 0.78602871,
        0.77263158, 0.77263158, 0.77358852, 0.77186603, 0.78717703,
        0.7722488 , 0.77244019, 0.77358852, 0.77110048, 0.78717703]),
 'split4_test_score': array([0.83253589, 0.8323445 , 0.8323445 , 0.82794258,
0.68382775,
        0.8323445 , 0.83215311, 0.83291866, 0.82698565, 0.68382775,
        0.83291866, 0.83291866, 0.83291866, 0.83406699, 0.68382775,
        0.8323445 , 0.8323445 , 0.83215311, 0.83425837, 0.68382775,
        0.83311005, 0.83311005, 0.83291866, 0.83157895, 0.68382775]),
 'mean_test_score': array([0.7953445 , 0.79530629, 0.7955337 , 0.79243187,
0.76143056,
```

```
        0.79358329, 0.79354509, 0.79350334, 0.79220039, 0.76539974,
        0.7932431 , 0.7932431 , 0.79312852, 0.79278427, 0.76559259,
        0.7926427 , 0.79264272, 0.79298111, 0.79212776, 0.76578356,
        0.79241531, 0.79245351, 0.79253174, 0.79023619, 0.76566898]),
 'std_test_score': array([0.02036348, 0.02022411, 0.02018156, 0.01897866,
0.04052587,
        0.0213952 , 0.02132602, 0.02159337, 0.01975809, 0.04252833,
        0.02178892, 0.02178892, 0.02176089, 0.02289312, 0.04279299,
        0.02182471, 0.02187083, 0.02151123, 0.02318506, 0.04277193,
        0.02218278, 0.02214816, 0.02195392, 0.02169364, 0.0427192 ]),
 'rank_test_score': array([ 2,  3,  1, 16, 25,  4,  5,  6, 18, 24,  7,  7,  9,
11, 23, 13, 12,
        10, 19, 21, 17, 15, 14, 20, 22], dtype=int32),
 'split0_train_score': array([0.84454324, 0.84453129, 0.84451934, 0.84407715,
0.82021129,
        0.8454037 , 0.84534395, 0.84517663, 0.84405325, 0.82016349,
        0.84521249, 0.84527224, 0.84520054, 0.84371863, 0.82013959,
        0.8454037 , 0.8454037 , 0.84516468, 0.84345571, 0.82015154,
        0.84579808, 0.84578613, 0.84563077, 0.84551126, 0.82016349]),
 'split1_train_score': array([0.83661842, 0.83660663, 0.83657124, 0.83612301,
0.82932871,
        0.83752669, 0.83755028, 0.83734975, 0.8361466 , 0.82947026,
        0.83824622, 0.83816365, 0.83814006, 0.83814006, 0.82945846,
        0.83844675, 0.83844675, 0.83855291, 0.83825802, 0.82939948,
        0.83877703, 0.83877703, 0.8388478 , 0.83798672, 0.82944667]),
 'split2_train_score': array([0.83319584, 0.8331487 , 0.83320763, 0.83279515,
0.82468711,
        0.83461004, 0.83461004, 0.83451576, 0.83458647, 0.82507601,
        0.83515214, 0.83515214, 0.83506965, 0.83457468, 0.82480496,
        0.83567068, 0.83569425, 0.83575318, 0.83485752, 0.82484031,
        0.83601244, 0.83603601, 0.83576496, 0.83286586, 0.82480496]),
 'split3_train_score': array([0.83880548, 0.83879369, 0.83897047, 0.83795696,
0.81188866,
        0.84006647, 0.84011361, 0.84010182, 0.83987791, 0.82909468,
        0.84050251, 0.84049073, 0.8405143 , 0.84056143, 0.82949537,
        0.84069107, 0.84070285, 0.84089141, 0.84017253, 0.82950715,
        0.84096212, 0.84095034, 0.84110354, 0.84083249, 0.82957786]),
 'split4_train_score': array([0.8224244 , 0.82241261, 0.8224244 , 0.82102199,
0.7459283 ,
        0.8228958 , 0.82288401, 0.82288401, 0.82143446, 0.7459283 ,
        0.82295472, 0.82295472, 0.82282509, 0.82250689, 0.7459283 ,
        0.82337898, 0.82337898, 0.82339077, 0.82233012, 0.7459283 ,
        0.82341434, 0.82343791, 0.82357932, 0.82323756, 0.7459283 ]),
 'mean_train_score': array([0.83511748, 0.83509858, 0.83513861, 0.83439485,
0.80640881,
        0.83610054, 0.83610038, 0.8360056 , 0.83521974, 0.80994655,
        0.83641362, 0.8364067 , 0.83634993, 0.83590034, 0.80996534,
```

```
        0.83671824, 0.83672531, 0.83675059, 0.83581478, 0.80996536,
        0.8369928 , 0.83699748, 0.83698528, 0.83608678, 0.80998426]),
 'std_train_score': array([0.00734174, 0.00734357, 0.00735001, 0.00762646,
0.03078262,
        0.00749531, 0.0074906 , 0.00744432, 0.0076292 , 0.03218524,
        0.0074864 , 0.00749525, 0.00752915, 0.00733498, 0.0322052 ,
        0.00739245, 0.00739305, 0.0073576 , 0.00729514, 0.03220352,
        0.00750743, 0.00749425, 0.00743481, 0.0076214 , 0.03221531])}
```

[85]:
```
lg2.best_estimator_
```

[85]:
```
LogisticRegression(C=0.5, class_weight=None, dual=False, fit_intercept=True,
                   intercept_scaling=1, l1_ratio=None, max_iter=100,
                   multi_class='warn', n_jobs=None, penalty='l2',
                   random_state=0, solver='warn', tol=0.0001, verbose=0,
                   warm_start=False)
```

[86]:
```
lg_pred2 = lg2.best_estimator_.predict(gu_Xtest)
lg_pred2
```

[86]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[87]:
```
prob6 = lg2.best_estimator_.predict_proba(gu_Xtest)
prob6
```

[87]:
```
array([[0.91372545, 0.08627455],
       [0.96230571, 0.03769429],
       [0.96092094, 0.03907906],
       …,
       [0.96151863, 0.03848137],
       [0.89257942, 0.10742058],
       [0.97582683, 0.02417317]])
```

[88]:
```
lg_matrix2 = metrics.confusion_matrix(gu_ytest, lg_pred2)
lg_matrix2
```

[88]:
```
array([[2038,   15],
       [ 116,   19]])
```

[89]:
```
target_names2 = ['Still alive at 30 day', 'Died in 30 days']
print("", classification_report(gu_ytest, lg_pred2, target_names=target_names2))
```

```
                        precision    recall  f1-score   support

Still alive at 30 day       0.95      0.99      0.97      2053
      Died in 30 days       0.56      0.14      0.22       135
```

```
    accuracy                           0.94      2188
   macro avg       0.75      0.57      0.60      2188
weighted avg       0.92      0.94      0.92      2188
```

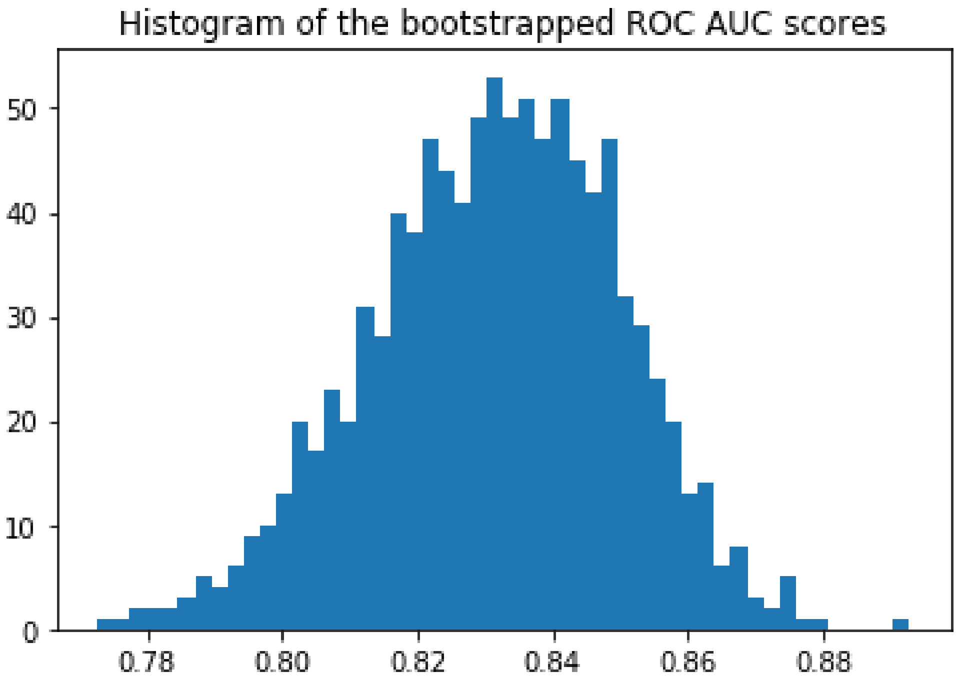
[90]:
```
lg_probs2 = lg2.best_estimator_.predict_proba(gu_Xtest)[:,1]
print(roc_auc_score(gu_ytest, lg_probs2))
```

```
0.8302033158341001
```

[91]:
```
#Boostrapping calculated 95% CI
gu_ytest = gu_ytest.values
y_pred = lg_probs2
y_true = gu_ytest

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
```

```
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.8302
```




```
Confidence interval for the score: [0.8000 - 0.8594]
```

[92]:

```
#pROC calculated 95% CI without bootstrapping
alpha = .95
gu_ytest = gu_ytest.reshape((2188,))
y_pred = lg_probs2
y_true = gu_ytest

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)
```

```
ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.8302033158341
AUC COV: 0.0003444411856651161
95% AUC CI: [0.7938281  0.86657854]
```

[ ]:

```

```

11 Ridge Regression Model

[93]:

```
from sklearn.linear_model import RidgeCV
ridgecv = RidgeCV(alphas=[1e-3, 1e-2, 1e-1, 1, 10], cv=5, fit_intercept=True,␣
↪scoring=None, normalize=False)
ridgecv=ridgecv.fit(gu_Xtrain,gu_ytrain)
gu_ridgecv = ridgecv.predict(gu_Xtest)
roc_auc_score(gu_ytest, gu_ridgecv)
```

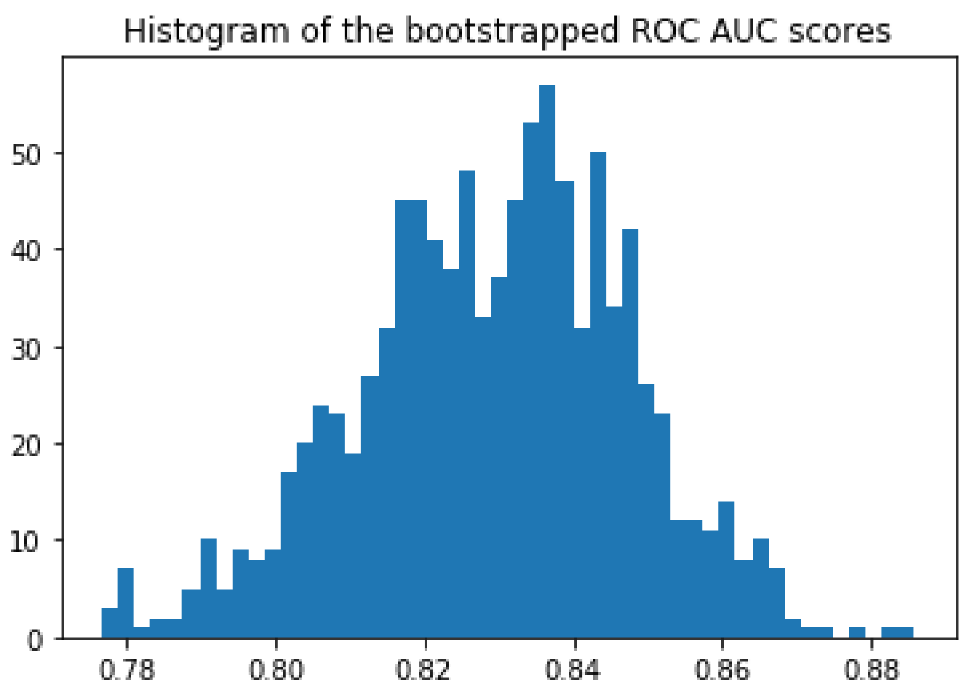
[93]: 0.8279590842669263

[94]:

```
#Boostrapping calculated 95% CI
y_pred = gu_ridgecv
y_true = gu_ytest

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))
```

```python
import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.8280
```




```
Confidence interval for the score: [0.7982 - 0.8586]
```

[95]:
```python
#pROC calculated 95% CI without bootstrapping
alpha = .95
gu_ridgecv = gu_ridgecv.reshape((2188,))
y_pred = gu_ridgecv
```

```
y_true = gu_ytest

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.8279590842669264
AUC COV: 0.0003523557006508896
95% AUC CI: [0.79116833 0.86474984]
```

[ ]:

```
```

12 Artificial Neural Network

[96]:

```
scaler = StandardScaler()
# Fit only to the training data
scaler = scaler.fit(gu_Xtrain)
gu_Xtrains = scaler.transform(gu_Xtrain)
gu_Xtests = scaler.transform(gu_Xtest)
```

[97]:

```
seed(1)
# skf = StratifiedKFold(n_splits=5)
params7 = {'alpha' : [0.0001,0.01],
           'power_t': [0.5,0.75],
           'max_iter': [200,250]}
mlp2 = MLPClassifier(solver='lbfgs', random_state=1)
mlp2 = GridSearchCV(mlp2, cv=5, param_grid=params7, scoring = 'roc_auc',refit =␣
 ↪True,
                    n_jobs=-1, verbose = 5, return_train_score=True)

mlp2.fit(gu_Xtrains, gu_ytrain)
```

```
mlp2.cv_results_
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  26 out of  40 | elapsed:    3.0s remaining:    1.6s
[Parallel(n_jobs=-1)]: Done  35 out of  40 | elapsed:    3.2s remaining:    0.5s
[Parallel(n_jobs=-1)]: Done  40 out of  40 | elapsed:    3.3s finished
/usr/local/lib/python3.7/site-
packages/sklearn/neural_network/multilayer_perceptron.py:921:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples, ), for example using
ravel().
  y = column_or_1d(y, warn=True)
```

[97]:
```
{'mean_fit_time': array([0.24871483, 0.30837779, 0.31661859, 0.32108974,
0.43691769,
        0.4199564 , 0.36917367, 0.28871627]),
 'std_fit_time': array([0.02404706, 0.02613244, 0.02350786, 0.02054395,
0.04495035,
        0.02794388, 0.00951518, 0.02044128]),
 'mean_score_time': array([0.00638633, 0.00591998, 0.00691061, 0.00545778,
0.00723748,
        0.00562401, 0.00423288, 0.00237317]),
 'std_score_time': array([0.00081622, 0.00028137, 0.00288459, 0.00023585,
0.00296454,
        0.00142962, 0.00136468, 0.00043304]),
 'param_alpha': masked_array(data=[0.0001, 0.0001, 0.0001, 0.0001, 0.01, 0.01,
0.01, 0.01],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_max_iter': masked_array(data=[200, 200, 250, 250, 200, 200, 250, 250],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'param_power_t': masked_array(data=[0.5, 0.75, 0.5, 0.75, 0.5, 0.75, 0.5,
0.75],
              mask=[False, False, False, False, False, False, False, False],
        fill_value='?',
             dtype=object),
 'params': [{'alpha': 0.0001, 'max_iter': 200, 'power_t': 0.5},
  {'alpha': 0.0001, 'max_iter': 200, 'power_t': 0.75},
  {'alpha': 0.0001, 'max_iter': 250, 'power_t': 0.5},
  {'alpha': 0.0001, 'max_iter': 250, 'power_t': 0.75},
  {'alpha': 0.01, 'max_iter': 200, 'power_t': 0.5},
  {'alpha': 0.01, 'max_iter': 200, 'power_t': 0.75},
  {'alpha': 0.01, 'max_iter': 250, 'power_t': 0.5},
```

```
 {'alpha': 0.01, 'max_iter': 250, 'power_t': 0.75}],
'split0_test_score': array([0.75896739, 0.75896739, 0.75896739, 0.75896739,
0.75126812,
        0.75126812, 0.75126812, 0.75126812]),
'split1_test_score': array([0.5891495 , 0.5891495 , 0.5891495 , 0.5891495 ,
0.57360793,
        0.57360793, 0.57360793, 0.57360793]),
'split2_test_score': array([0.74717703, 0.74717703, 0.74717703, 0.74717703,
0.73741627,
        0.73741627, 0.73741627, 0.73741627]),
'split3_test_score': array([0.6415311 , 0.6415311 , 0.6415311 , 0.6415311 ,
0.66660287,
        0.66660287, 0.66660287, 0.66660287]),
'split4_test_score': array([0.74220096, 0.74220096, 0.74220096, 0.74220096,
0.74411483,
        0.74411483, 0.74411483, 0.74411483]),
'mean_test_score': array([0.69581855, 0.69581855, 0.69581855, 0.69581855,
0.6945968 ,
        0.6945968 , 0.6945968 , 0.6945968 ]),
'std_test_score': array([0.06800126, 0.06800126, 0.06800126, 0.06800126,
0.06774062,
        0.06774062, 0.06774062, 0.06774062]),
'rank_test_score': array([1, 1, 1, 1, 5, 5, 5, 5], dtype=int32),
'split0_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
'split1_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
'split2_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
'split3_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
'split4_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
'mean_train_score': array([1., 1., 1., 1., 1., 1., 1., 1.]),
'std_train_score': array([0., 0., 0., 0., 0., 0., 0., 0.])}
```

[98]:
```
mlp2.best_estimator_
```

[98]:
```
MLPClassifier(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
              beta_2=0.999, early_stopping=False, epsilon=1e-08,
              hidden_layer_sizes=(100,), learning_rate='constant',
              learning_rate_init=0.001, max_iter=200, momentum=0.9,
              n_iter_no_change=10, nesterovs_momentum=True, power_t=0.5,
              random_state=1, shuffle=True, solver='lbfgs', tol=0.0001,
              validation_fraction=0.1, verbose=False, warm_start=False)
```

[99]:
```
ann_pred2 = mlp2.best_estimator_.predict(gu_Xtests)
ann_pred2
```

[99]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[100]:
```
prob7 = mlp2.best_estimator_.predict_proba(gu_Xtests)
prob7
```

[100]:
```
array([[1.00000000e+00, 4.92987998e-92],
       [1.00000000e+00, 3.30660487e-78],
       [1.00000000e+00, 5.34809233e-83],
       …,
       [1.00000000e+00, 1.21594736e-30],
       [1.00000000e+00, 8.23828743e-48],
       [1.00000000e+00, 8.12154336e-39]])
```

[101]:
```
ann_matrix2 = metrics.confusion_matrix(gu_ytest, ann_pred2)
ann_matrix2
```

[101]:
```
array([[1947,  106],
       [  93,   42]])
```

[102]:
```
target_names2 = ['Still alive at 30 day', 'Died in 30 days']
print("", classification_report(gu_ytest, ann_pred2,
                                target_names=target_names2))
```

```
                        precision    recall  f1-score   support

Still alive at 30 day       0.95      0.95      0.95      2053
      Died in 30 days       0.28      0.31      0.30       135

             accuracy                           0.91      2188
            macro avg       0.62      0.63      0.62      2188
         weighted avg       0.91      0.91      0.91      2188
```

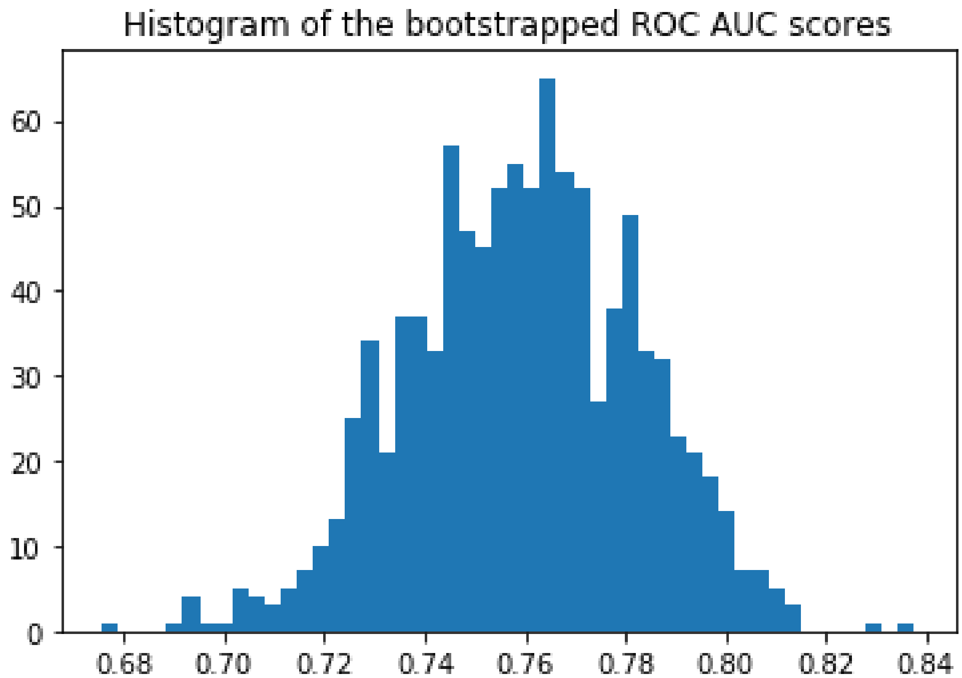
[103]:
```
ann_probs2 = mlp2.best_estimator_.predict_proba(gu_Xtests)[:,1]
print(roc_auc_score(gu_ytest, ann_probs2))
```

```
0.759576771120853
```

[104]:
```
#Boostrapping calculated 95% CI
y_pred = ann_probs2
y_true = gu_ytest

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
```

```
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.7596
```




```
Confidence interval for the score: [0.7232 - 0.7963]
```

[105]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
y_pred = ann_probs2
y_true = gu_ytest

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.759576771120853
AUC COV: 0.0005295762772071846
95% AUC CI: [0.71447305 0.80468049]
```

[ ]:

13 Random Forest

[106]:
```
seed(42)

params8 = {'n_estimators' : [10,100,150],
           'min_samples_leaf': [1,2,3]}
rf2 = RandomForestClassifier(random_state=42)
rf2 = GridSearchCV(rf2, cv=5, param_grid=params8, scoring = 'roc_auc',refit =␣
 ↪True,
                   n_jobs=-1, verbose = 5, return_train_score=True)

rf2.fit(gu_Xtrain,gu_ytrain)
rf2.cv_results_
```

```
Fitting 5 folds for each of 9 candidates, totalling 45 fits
```

```
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  32 out of  45 | elapsed:    3.1s remaining:    1.2s
[Parallel(n_jobs=-1)]: Done  42 out of  45 | elapsed:    3.3s remaining:    0.2s
[Parallel(n_jobs=-1)]: Done  45 out of  45 | elapsed:    3.4s finished
/usr/local/lib/python3.7/site-packages/sklearn/model_selection/_search.py:715:
DataConversionWarning: A column-vector y was passed when a 1d array was
expected. Please change the shape of y to (n_samples,), for example using
ravel().
  self.best_estimator_.fit(X, y, **fit_params)
```

[106]:
```
{'mean_fit_time': array([0.0431448 , 0.31748538, 0.47422519, 0.03773494,
0.31329279,
        0.48183465, 0.03696904, 0.31390924, 0.33778834]),
 'std_fit_time': array([0.00311095, 0.00492449, 0.00975234, 0.00180364,
0.01112705,
        0.00534404, 0.00012415, 0.00353635, 0.04553103]),
 'mean_score_time': array([0.00668821, 0.02331481, 0.03344364, 0.00601273,
0.02352328,
        0.03404474, 0.0068069 , 0.02332582, 0.01948948]),
 'std_score_time': array([0.00034828, 0.00066868, 0.00229864, 0.00027037,
0.00032041,
        0.00068371, 0.00073997, 0.00156654, 0.00032613]),
 'param_min_samples_leaf': masked_array(data=[1, 1, 1, 2, 2, 2, 3, 3, 3],
              mask=[False, False, False, False, False, False, False, False,
```

```
                    False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[10, 100, 150, 10, 100, 150, 10, 100,
150],
              mask=[False, False, False, False, False, False, False, False,
                    False],
        fill_value='?',
             dtype=object),
 'params': [{'min_samples_leaf': 1, 'n_estimators': 10},
  {'min_samples_leaf': 1, 'n_estimators': 100},
  {'min_samples_leaf': 1, 'n_estimators': 150},
  {'min_samples_leaf': 2, 'n_estimators': 10},
  {'min_samples_leaf': 2, 'n_estimators': 100},
  {'min_samples_leaf': 2, 'n_estimators': 150},
  {'min_samples_leaf': 3, 'n_estimators': 10},
  {'min_samples_leaf': 3, 'n_estimators': 100},
  {'min_samples_leaf': 3, 'n_estimators': 150}],
 'split0_test_score': array([0.73632246, 0.81757246, 0.81666667, 0.69248188,
0.79981884,
        0.81322464, 0.77753623, 0.83115942, 0.83822464]),
 'split1_test_score': array([0.72053776, 0.73951182, 0.74094203, 0.63567887,
0.75629291,
        0.74980931, 0.70061022, 0.76735317, 0.76372998]),
 'split2_test_score': array([0.71406699, 0.79282297, 0.79110048, 0.78593301,
0.77358852,
        0.77339713, 0.73645933, 0.78832536, 0.79406699]),
 'split3_test_score': array([0.69253589, 0.76870813, 0.7569378 , 0.7508134 ,
0.77521531,
        0.76669856, 0.69416268, 0.76669856, 0.75923445]),
 'split4_test_score': array([0.85339713, 0.8476555 , 0.85090909, 0.74277512,
0.82593301,
        0.8430622 , 0.83732057, 0.84976077, 0.85550239]),
 'mean_test_score': array([0.74334697, 0.79325071, 0.79131144, 0.72143872,
0.78616797,
        0.78924417, 0.74922326, 0.80067826, 0.80217458]),
 'std_test_score': array([0.05672565, 0.03752678, 0.03977511, 0.05229473,
0.02423484,
        0.03400995, 0.05311361, 0.0339244 , 0.03880603]),
 'rank_test_score': array([8, 3, 4, 9, 6, 5, 7, 2, 1], dtype=int32),
 'split0_train_score': array([0.99986257, 1.        , 1.        , 0.99442492,
0.99960562,
        0.99955782, 0.98568287, 0.99600841, 0.99709594]),
 'split1_train_score': array([0.99840169, 1.        , 1.        , 0.9910471 ,
0.99961074,
        0.9997169 , 0.9869658 , 0.99639053, 0.99689774]),
 'split2_train_score': array([0.99987037, 1.        , 1.        , 0.9958517 ,
```

```
0.99892757,
        0.99923398, 0.98869823, 0.99629953, 0.99648809]),
 'split3_train_score': array([0.99989983, 1.        , 1.        , 0.99582224,
0.99926933,
        0.9992929 , 0.98688925, 0.99560421, 0.99604026]),
 'split4_train_score': array([0.99975252, 1.        , 1.        , 0.99544512,
0.99951682,
        0.99963467, 0.98635303, 0.99675914, 0.99698305]),
 'mean_train_score': array([0.99955739, 1.        , 1.        , 0.99451822,
0.99938602,
        0.99948725, 0.98691784, 0.99621236, 0.99670102]),
 'std_train_score': array([5.80011721e-04, 4.96506831e-17, 0.00000000e+00,
1.81080856e-03,
        2.60624598e-04, 1.90457287e-04, 1.00170650e-03, 3.87232642e-04,
        3.88884105e-04])}
```

[107]:
```
rf2.best_estimator_
```

[107]:
```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                       max_depth=None, max_features='auto', max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=3, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=150,
                       n_jobs=None, oob_score=False, random_state=42, verbose=0,
                       warm_start=False)
```

[108]:
```
rf_pred2 = rf2.best_estimator_.predict(gu_Xtest)
rf_pred2
```

[108]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[109]:
```
prob8 = rf2.best_estimator_.predict_proba(gu_Xtest)
prob8
```

[109]:
```
array([[0.96881419, 0.03118581],
       [0.94229728, 0.05770272],
       [0.99634343, 0.00365657],
       …,
       [0.97513973, 0.02486027],
       [0.83568952, 0.16431048],
       [0.97687497, 0.02312503]])
```

[110]:
```
rf_matrix2 = metrics.confusion_matrix(gu_ytest, rf_pred2)
rf_matrix2
```

[110]:
```
array([[2051,    2],
       [ 128,    7]])
```

[111]:
```
print("", classification_report(gu_ytest, rf_pred2,
                                target_names=target_names2))
```

```
                     precision    recall  f1-score   support

Still alive at 30 day       0.94      1.00      0.97      2053
      Died in 30 days       0.78      0.05      0.10       135

             accuracy                           0.94      2188
            macro avg       0.86      0.53      0.53      2188
         weighted avg       0.93      0.94      0.92      2188
```

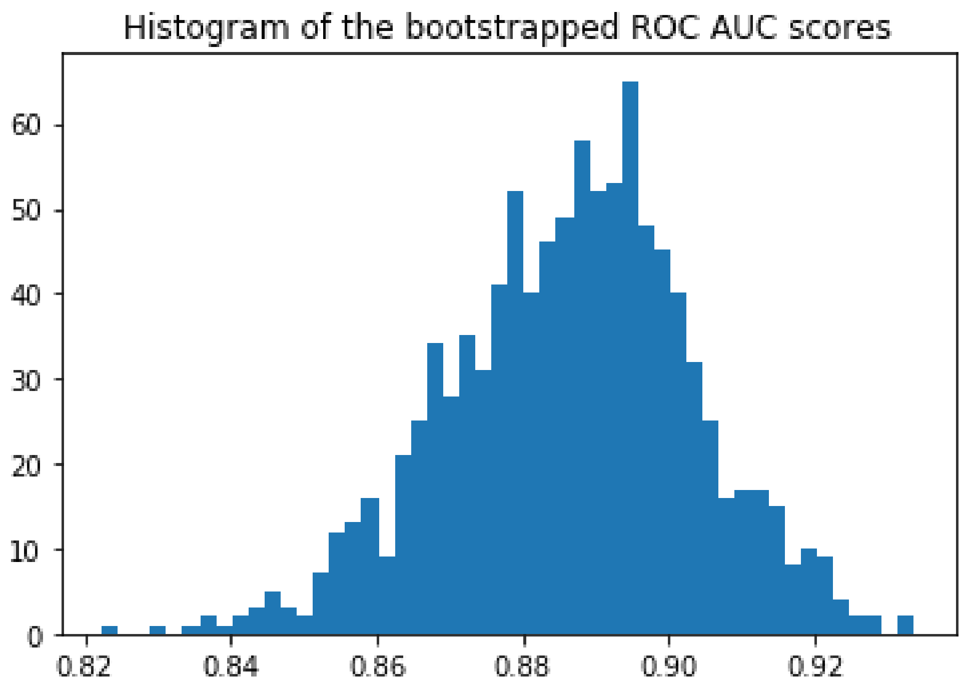
[112]:
```
rf_probs2 = rf2.best_estimator_.predict_proba(gu_Xtest)[:,1]
print(roc_auc_score(gu_ytest, rf_probs2))
```

```
0.885331312803305
```

[113]:
```
#Boostrapping calculated 95% CI
y_pred = rf_probs2
y_true = gu_ytest

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
```

```
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.8853
```




```
Confidence interval for the score: [0.8575 - 0.9138]
```

[114]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
y_pred = rf_probs2
y_true = gu_ytest

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)
```

```
ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.885331312803305
AUC COV: 0.0002835084573262921
95% AUC CI: [0.85233001 0.91833262]
```

[ ]:

14 Gradient Boosting Machines

[115]:

```
seed(10)

params9 = {'learning_rate' : [0.05,0.1,0.2],
           'n_estimators': [100,150],
           'min_samples_split': [2,3,4]}
gbm2 = GradientBoostingClassifier(random_state=10)
gbm2 = GridSearchCV(gbm2, cv=5, param_grid=params9, scoring = 'roc_auc',refit =␣
 ↪True,
                    n_jobs=-1, verbose = 5, return_train_score=True)

gbm2.fit(gu_Xtrain,gu_ytrain)
gbm2.cv_results_
```

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent workers.
[Parallel(n_jobs=-1)]: Done  48 tasks      | elapsed:    3.5s
[Parallel(n_jobs=-1)]: Done  86 out of  90 | elapsed:    4.4s remaining:    0.2s
[Parallel(n_jobs=-1)]: Done  90 out of  90 | elapsed:    4.5s finished
/usr/local/lib/python3.7/site-
packages/sklearn/ensemble/gradient_boosting.py:1450: DataConversionWarning: A
column-vector y was passed when a 1d array was expected. Please change the shape
of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

[115]:
```
{'mean_fit_time': array([0.22256212, 0.34172802, 0.23362408, 0.35020294,
0.22939553,
        0.35094404, 0.23809605, 0.36700315, 0.24748402, 0.36184278,
        0.24461427, 0.37012062, 0.25583777, 0.38549562, 0.25356526,
        0.36602039, 0.24349747, 0.2658812 ]),
 'std_fit_time': array([0.00385851, 0.00451759, 0.00361519, 0.00726972,
0.00304873,
        0.00746689, 0.00321686, 0.00459206, 0.00859492, 0.00679286,
        0.00589788, 0.01086037, 0.00651552, 0.00786239, 0.00857925,
        0.01194572, 0.0051395 , 0.02943227]),
 'mean_score_time': array([0.00541196, 0.00759826, 0.00531902, 0.00563464,
0.0054883 ,
        0.00552077, 0.00528045, 0.0055562 , 0.00560541, 0.00684323,
        0.00533257, 0.00595651, 0.00610633, 0.00580502, 0.00594573,
        0.00556779, 0.00638723, 0.00353961]),
 'std_score_time': array([0.00019735, 0.00059642, 0.00021568, 0.00039022,
0.00071267,
        0.00013416, 0.00019982, 0.00010537, 0.00042535, 0.00228823,
        0.00014897, 0.00012158, 0.0014703 , 0.00016364, 0.0009264 ,
        0.00105673, 0.00156847, 0.00019318]),
 'param_learning_rate': masked_array(data=[0.05, 0.05, 0.05, 0.05, 0.05, 0.05,
0.1, 0.1, 0.1, 0.1,
                    0.1, 0.1, 0.2, 0.2, 0.2, 0.2, 0.2, 0.2],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_min_samples_split': masked_array(data=[2, 2, 3, 3, 4, 4, 2, 2, 3, 3, 4,
4, 2, 2, 3, 3, 4, 4],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'param_n_estimators': masked_array(data=[100, 150, 100, 150, 100, 150, 100,
150, 100, 150, 100,
                    150, 100, 150, 100, 150, 100, 150],
              mask=[False, False, False, False, False, False, False, False,
                    False, False, False, False, False, False, False, False,
                    False, False],
        fill_value='?',
             dtype=object),
 'params': [{'learning_rate': 0.05,
   'min_samples_split': 2,
   'n_estimators': 100},
  {'learning_rate': 0.05, 'min_samples_split': 2, 'n_estimators': 150},
```

```
  {'learning_rate': 0.05, 'min_samples_split': 3, 'n_estimators': 100},
  {'learning_rate': 0.05, 'min_samples_split': 3, 'n_estimators': 150},
  {'learning_rate': 0.05, 'min_samples_split': 4, 'n_estimators': 100},
  {'learning_rate': 0.05, 'min_samples_split': 4, 'n_estimators': 150},
  {'learning_rate': 0.1, 'min_samples_split': 2, 'n_estimators': 100},
  {'learning_rate': 0.1, 'min_samples_split': 2, 'n_estimators': 150},
  {'learning_rate': 0.1, 'min_samples_split': 3, 'n_estimators': 100},
  {'learning_rate': 0.1, 'min_samples_split': 3, 'n_estimators': 150},
  {'learning_rate': 0.1, 'min_samples_split': 4, 'n_estimators': 100},
  {'learning_rate': 0.1, 'min_samples_split': 4, 'n_estimators': 150},
  {'learning_rate': 0.2, 'min_samples_split': 2, 'n_estimators': 100},
  {'learning_rate': 0.2, 'min_samples_split': 2, 'n_estimators': 150},
  {'learning_rate': 0.2, 'min_samples_split': 3, 'n_estimators': 100},
  {'learning_rate': 0.2, 'min_samples_split': 3, 'n_estimators': 150},
  {'learning_rate': 0.2, 'min_samples_split': 4, 'n_estimators': 100},
  {'learning_rate': 0.2, 'min_samples_split': 4, 'n_estimators': 150}],
 'split0_test_score': array([0.77382246, 0.74565217, 0.78623188, 0.77028986,
0.76449275,
        0.75199275, 0.76539855, 0.75742754, 0.76105072, 0.7509058 ,
        0.74873188, 0.74764493, 0.75797101, 0.75181159, 0.77952899,
        0.77663043, 0.7365942 , 0.71050725]),
 'split1_test_score': array([0.73455378, 0.74523265, 0.74780702, 0.74599542,
0.74885584,
        0.75076278, 0.74084668, 0.73455378, 0.73741419, 0.70747521,
        0.73779558, 0.7305492 , 0.73512586, 0.71720061, 0.70175439,
        0.67982456, 0.71929825, 0.69012204]),
 'split2_test_score': array([0.77205742, 0.77301435, 0.77148325, 0.77799043,
0.77186603,
        0.77875598, 0.78373206, 0.78411483, 0.77952153, 0.78009569,
        0.78296651, 0.79043062, 0.78047847, 0.78028708, 0.78392344,
        0.78354067, 0.76574163, 0.74392344]),
 'split3_test_score': array([0.71100478, 0.68956938, 0.70258373, 0.6815311 ,
0.72114833,
        0.68660287, 0.64210526, 0.65129187, 0.63751196, 0.6845933 ,
        0.62411483, 0.63559809, 0.6076555 , 0.5691866 , 0.58660287,
        0.5984689 , 0.62602871, 0.58947368]),
 'split4_test_score': array([0.82755981, 0.81818182, 0.82526316, 0.82315789,
0.82870813,
        0.81952153, 0.81913876, 0.8061244 , 0.82411483, 0.83062201,
        0.81741627, 0.81033493, 0.82277512, 0.79177033, 0.82507177,
        0.8292823 , 0.8262201 , 0.81722488]),
 'mean_test_score': array([0.7637934 , 0.75431212, 0.76668756, 0.75979783,
0.76699846,
        0.75751508, 0.75025846, 0.7467088 , 0.74793334, 0.75070926,
        0.74221088, 0.74290959, 0.74082065, 0.72208836, 0.73541342,
        0.73357139, 0.73476854, 0.71023694]),
 'std_test_score': array([0.03965896, 0.04185947, 0.04071548, 0.04637067,
```

```
0.0353775 ,
        0.04331702, 0.05974167, 0.05342025, 0.06203115, 0.05190187,
        0.06524184, 0.06075918, 0.07252206, 0.08058388, 0.08433274,
        0.08319232, 0.06534065, 0.07418203]),
 'rank_test_score': array([ 3,  6,  2,  4,  1,  5,  8, 10,  9,  7, 12, 11, 13,
17, 14, 16, 15,
        18], dtype=int32),
 'split0_train_score': array([0.96441632, 0.97920551, 0.96136288, 0.98126105,
0.96078923,
        0.98096228, 0.98704527, 0.99378555, 0.98962666, 0.99580525,
        0.98771452, 0.99539892, 0.99916344, 1.        , 0.99998805,
        1.        , 0.99939051, 1.        ]),
 'split1_train_score': array([0.96179978, 0.9758425 , 0.9625606 , 0.9752999 ,
0.95852059,
        0.97386083, 0.99284004, 0.99933944, 0.98685964, 0.99902096,
        0.99203793, 0.99542329, 0.99911533, 1.        , 1.        ,
        1.        , 1.        , 1.        ]),
 'split2_train_score': array([0.94661418, 0.97316567, 0.94852335, 0.9745563 ,
0.94762769,
        0.97388455, 0.98368963, 0.99614632, 0.98559879, 0.99629953,
        0.98736064, 0.99589884, 0.99985858, 1.        , 0.99981144,
        1.        , 0.99941075, 1.        ]),
 'split3_train_score': array([0.95035001, 0.97647135, 0.94496429, 0.97045513,
0.95118085,
        0.97113866, 0.9809732 , 0.99402503, 0.98148585, 0.99641738,
        0.98154477, 0.99252834, 1.        , 1.        , 0.99868009,
        1.        , 0.9985151 , 1.        ]),
 'split4_train_score': array([0.94395668, 0.96426804, 0.94526481, 0.96371414,
0.94537087,
        0.96552903, 0.97766752, 0.99895114, 0.98188653, 0.99645273,
        0.98304146, 0.99435501, 0.99545101, 1.        , 0.99949325,
        1.        , 0.99996465, 1.        ]),
 'mean_train_score': array([0.9534274 , 0.97379061, 0.95253518, 0.97305731,
0.95269785,
        0.97307507, 0.98444313, 0.9964495 , 0.98509149, 0.99679917,
        0.98633987, 0.99472088, 0.99871767, 1.        , 0.99959456,
        1.        , 0.9994562 , 1.        ]),
 'std_train_score': array([0.0082029 , 0.00513396, 0.00780652, 0.00580764,
0.00601776,
        0.00498344, 0.00521145, 0.00235272, 0.00307323, 0.00113487,
        0.00372215, 0.0012068 , 0.00167176, 0.        , 0.0004925 ,
        0.        , 0.00053783, 0.        ])}
```

[116]:

```
gbm2.best_estimator_
```

[116]:

```
GradientBoostingClassifier(criterion='friedman_mse', init=None,
                           learning_rate=0.05, loss='deviance', max_depth=3,
```

```
                                        max_features=None, max_leaf_nodes=None,
                                        min_impurity_decrease=0.0, min_impurity_split=None,
                                        min_samples_leaf=1, min_samples_split=4,
                                        min_weight_fraction_leaf=0.0, n_estimators=100,
                                        n_iter_no_change=None, presort='auto',
                                        random_state=10, subsample=1.0, tol=0.0001,
                                        validation_fraction=0.1, verbose=0,
                                        warm_start=False)
```

[117]:
```
gbm_pred2 = gbm2.best_estimator_.predict(gu_Xtest)
gbm_pred2
```

[117]:
```
array([0, 0, 0, …, 0, 0, 0])
```

[118]:
```
prob9 = gbm2.best_estimator_.predict_proba(gu_Xtest)
prob9
```

[118]:
```
array([[0.947186  , 0.052814  ],
       [0.97310495, 0.02689505],
       [0.97898104, 0.02101896],
       …,
       [0.9711283 , 0.0288717 ],
       [0.87880754, 0.12119246],
       [0.98384301, 0.01615699]])
```

[119]:
```
gbm_matrix2 = metrics.confusion_matrix(gu_ytest, gbm_pred2)
gbm_matrix2
```

[119]:
```
array([[2044,    9],
       [ 113,   22]])
```

[120]:
```
print("", classification_report(gu_ytest, gbm_pred2,
                                target_names=target_names2))
```

```
                        precision    recall  f1-score   support

Still alive at 30 day       0.95      1.00      0.97      2053
      Died in 30 days       0.71      0.16      0.27       135

             accuracy                           0.94      2188
            macro avg       0.83      0.58      0.62      2188
         weighted avg       0.93      0.94      0.93      2188
```

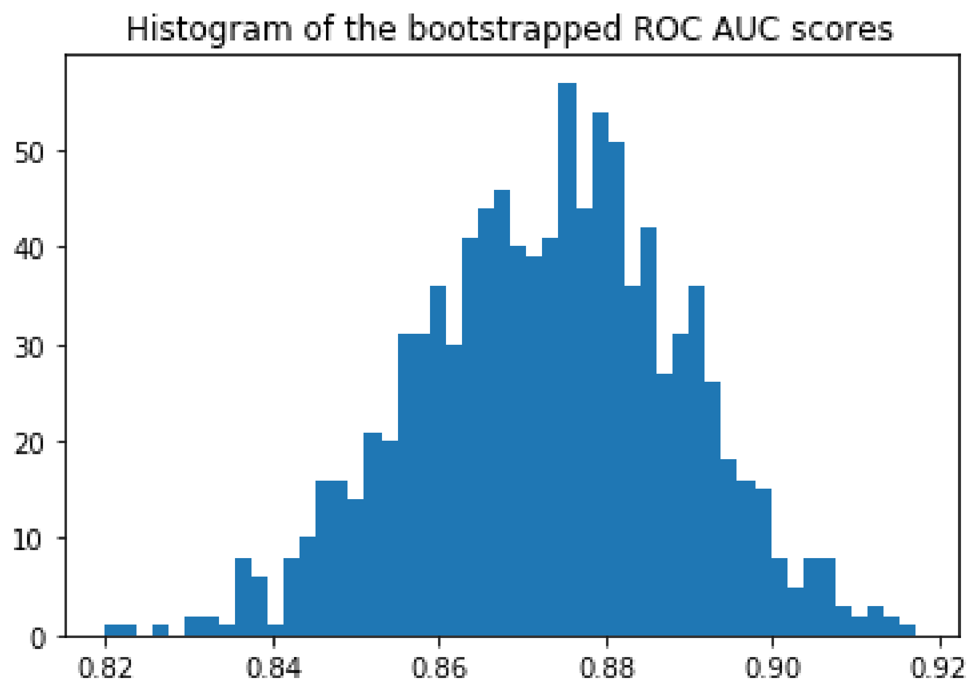
[121]:
```
gbm_probs2 = gbm2.best_estimator_.predict_proba(gu_Xtest)[:,1]
print(roc_auc_score(gu_ytest, gbm_probs2))
```

```
0.8720806047157728
```

[122]:

```python
#Boostrapping calculated 95% CI
y_pred = gbm_probs2
y_true = gu_ytest

print("Original ROC area: {:0.4f}".format(roc_auc_score(y_true, y_pred)))

n_bootstraps = 1000
rng_seed = 42  # control reproducibility
bootstrapped_scores = []

rng = np.random.RandomState(rng_seed)
for i in range(n_bootstraps):
    # bootstrap by sampling with replacement on the prediction indices
    indices = rng.randint(0, len(y_pred), len(y_pred))
    if len(np.unique(y_true[indices])) < 2:
        # We need at least one positive and one negative sample for ROC AUC
        # to be defined: reject the sample
        continue

    score = roc_auc_score(y_true[indices], y_pred[indices])
    bootstrapped_scores.append(score)
    #print("Bootstrap #{} ROC area: {:0.3f}".format(i + 1, score))

import matplotlib.pyplot as plt
plt.hist(bootstrapped_scores, bins=50)
plt.title('Histogram of the bootstrapped ROC AUC scores')
plt.show()

sorted_scores = np.array(bootstrapped_scores)
sorted_scores.sort()

# Computing the lower and upper bound of the 90% confidence interval
# You can change the bounds percentiles to 0.025 and 0.975 to get
# a 95% confidence interval instead.
confidence_lower = sorted_scores[int(0.05 * len(sorted_scores))]
confidence_upper = sorted_scores[int(0.95 * len(sorted_scores))]
print("Confidence interval for the score: [{:0.4f} - {:0.4}]".format(
    confidence_lower, confidence_upper))
```

```
Original ROC area: 0.8721
```




```
Confidence interval for the score: [0.8468 - 0.8985]
```

[123]:
```
#pROC calculated 95% CI without bootstrapping
alpha = .95
y_pred = gbm_probs2
y_true = gu_ytest

auc, auc_cov = delong_roc_variance(
    y_true,
    y_pred)

auc_std = np.sqrt(auc_cov)
lower_upper_q = np.abs(np.array([0, 1]) - (1 - alpha) / 2)

ci = stats.norm.ppf(
    lower_upper_q,
    loc=auc,
    scale=auc_std)

ci[ci > 1] = 1

print('AUC:', auc)
print('AUC COV:', auc_cov)
print('95% AUC CI:', ci)
```

```
AUC: 0.8720806047157728
AUC COV: 0.000262844614256567
95% AUC CI: [0.84030472 0.90385649]
```

[ ]:

```

```

[ ]:

```

```

[ ]:

```

```