Hw 1

Gadiel Eitan 211879051
Manar Awida 207364332

Exploratory Data Analysis

30000 patients: for each one, we have 40 features collected during his stay in the hospital ICU. 8 of them are vital signs of Sepsis, 26 are laboratory values (lab test result), and 6 are demographic/administrative values. Before starting, we'll extract and slice the data:

In [1]:
```
import os
import shutil
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from Analysis.utils import load_data, slice_data, concat_data # our own utils file, implemantation on github
```

In [2]:
```
load_data(tar_file='data.tar')
os.mkdir('Data/sliced')
# Train:
os.mkdir('Data/sliced/train')
slice_data('Data/raw/train', 'Data/sliced/train')
shutil.move('Data/sliced/train/labels.csv', 'Data/sliced/train_labels.csv')
# Test:
os.mkdir('Data/sliced/test')
slice_data('Data/raw/test', 'Data/sliced/test')
shutil.move('Data/sliced/test/labels.csv', 'Data/sliced/test_labels.csv')
```

```
Progress: 100%|██████████| 20000/20000 [01:09<00:00, 289.26it/s]
Progress: 100%|██████████| 10000/10000 [00:34<00:00, 291.82it/s]
```

Out[2]:
```
'Data/sliced/test_labels.csv'
```

In [3]:
```
train_df = concat_data(src_dir='Data/sliced/train', dst_dir='Data/sliced', name='train')
```

```
100%|██████████| 20000/20000 [00:41<00:00, 482.97it/s]
```

In [4]:
```
train_df.head(3)
```

Out[4]:

	HR	O2Sat	Temp	SBP	MAP	DBP	Resp	EtCO2	BaseExcess	HCO3	...	WBC	Fibrinogen	Platelets	Age	Gender	Unit1	Unit2	HospAdmTime	ICULOS	Patient
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	77.0	0	NaN	NaN	-679.93	1	100
1	58.5	96.0	NaN	133.0	54.0	30.0	28.5	NaN	NaN	NaN	...	NaN	NaN	NaN	77.0	0	NaN	NaN	-679.93	2	100
2	54.0	97.0	36.4	128.0	58.0	36.0	23.0	NaN	NaN	NaN	...	NaN	NaN	NaN	77.0	0	NaN	NaN	-679.93	3	100

3 rows × 41 columns

`train_df` is our full training set (20000 patients). Since the distribution of the data assumed to be real, we will analyze only the training set.

In [8]:
```
cols = train_df.columns
vital, lab, administrative = cols[:8], cols[8:34], cols[35:]
print("Vital:\t\t\t{}".format(list(vital)))
print("Lab test results:\t{}".format(list(lab)))
print("Administrative:\t\t{}".format(list(administrative)))
```

```
Vital:                  ['HR', 'O2Sat', 'Temp', 'SBP', 'MAP', 'DBP', 'Resp', 'EtCO2']
Lab test results:       ['BaseExcess', 'HCO3', 'FiO2', 'pH', 'PaCO2', 'SaO2', 'AST', 'BUN', 'Alkalinephos', 'Calcium', 'Chloride', 'Creatinine',
'Bilirubin_direct', 'Glucose', 'Lactate', 'Magnesium', 'Phosphate', 'Potassium', 'Bilirubin_total', 'TroponinI', 'Hct', 'Hgb', 'PTT', 'WBC', 'Fi
brinogen', 'Platelets']
Administrative:         ['Gender', 'Unit1', 'Unit2', 'HospAdmTime', 'ICULOS', 'PatientID']
```

Imbalance:

We experiment with a highly imbalanced dataset. Load train labels we save above:

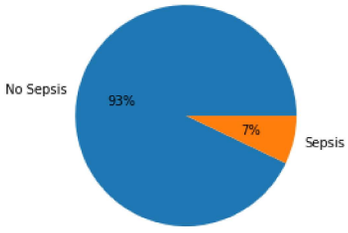
In [9]:
```
labels = pd.read_csv('Data/sliced/train_labels.csv', index_col=0) # labels.loc[id: int] returns the label of patient_id.psv
```

The ratio of the first class (`SepsisLabel == 1` or positive) to the second class (`SepsisLabel == 0` or negative) instances is:

In [10]:
```
positive_size = len(labels.loc[labels['label'] == 1])
negative_size = len(labels.loc[labels['label'] == 0])
print('Ratio: {:.2f} : 1.'.format(negative_size/positive_size, positive_size))
```

```
Ratio: 13.13 : 1.
```

In [11]:
```
negative_percent = negative_size/len(labels.index)
positive_percent = positive_size/len(labels.index)
titles = ['No Sepsis', 'Sepsis']
percents = [negative_percent, positive_percent]
fig, ax = plt.subplots()
ax.pie(percents, labels=titles, autopct='%1.0f%%')
plt.show()
```



Based on the plot above, we may use some resampling methods later.

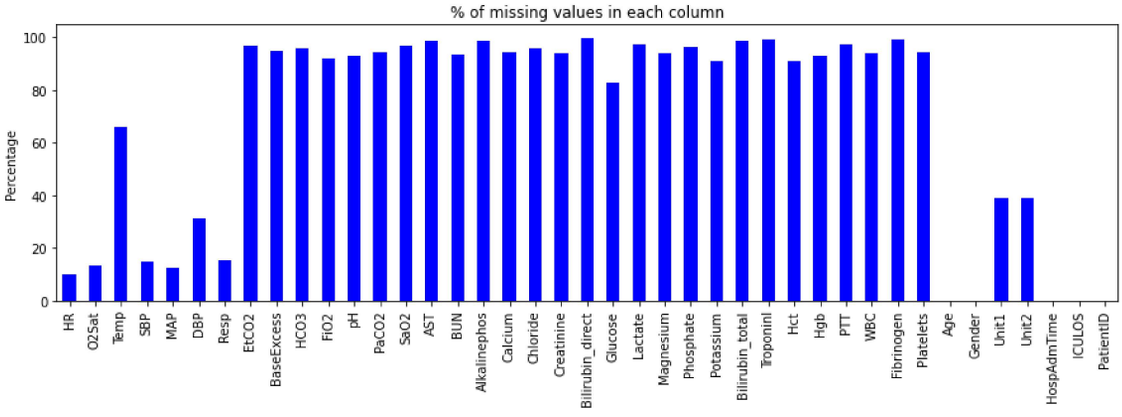
Missing data:

Percentage of missing values in each column:

In [12]:
```
missing = train_df.isnull().sum()*100/len(train_df.index)
missing
```

Out[12]:
```
HR                 9.952611
O2Sat             13.120266
Temp              66.091251
SBP               14.710585
MAP               12.551122
DBP               31.222402
Resp              15.460834
EtCO2             96.539956
BaseExcess        94.592597
HCO3              95.882446
FiO2              91.843321
pH                93.096075
PaCO2             94.488598
SaO2              96.547905
AST               98.430481
BUN               93.185898
Alkalinephos      98.447173
Calcium           94.189320
Chloride          95.525670
Creatinine        93.950984
Bilirubin_direct  99.815319
Glucose           82.850280
Lactate           97.391421
Magnesium         93.747491
Phosphate         96.056660
Potassium         90.725169
Bilirubin_total   98.551834
TroponinI         99.035925
Hct               91.158519
Hgb               92.644177
PTT               97.114797
WBC               93.637266
Fibrinogen        99.365277
Platelets         94.102014
Age                0.000000
Gender             0.000000
Unit1             38.910491
Unit2             38.910491
HospAdmTime        0.000000
ICULOS             0.000000
PatientID          0.000000
dtype: float64
```

In [16]:
```
import matplotlib.pyplot as plt
missing.plot.bar(figsize=(15,4), color='Blue')
plt.title("% of missing values in each column")
plt.ylabel('Percentage')
plt.show()
```



It seems that most of the laboratory values are missing, we might use it later to decide which columns are useful and bring any value. The vital sign EtCO2 is missing in 96.54% of the rows (yet still vital). End-Tidal CO2 is the amount of carbon dioxide (CO2) in exhaled air, which assesses ventilation. So a high ETCO2 is a good sign of good ventilation, while low ETCO2 is bad sign that represents hypoventilation. As expected, the age and the gender (demographic) are known for all the patients (no missing data). From this observation only, the following list may be our chosen feature set:

Analysis:

Some important stats:

In [20]:
```
train_stats = train_df.describe()
train_stats.T.head() # Example:
```

Out[20]:

	count	mean	std	min	25%	50%	75%	max
HR	679693.0	84.558897	17.344198	20.0	72.0	83.0	95.5	280.0
O2Sat	655783.0	97.196002	2.944657	20.0	96.0	98.0	99.5	100.0
Temp	255949.0	36.972567	0.770627	20.9	36.5	37.0	37.5	50.0
SBP	643779.0	123.550546	23.130050	20.0	107.0	121.0	138.0	299.0
MAP	660079.0	82.335320	16.282437	20.0	71.0	80.0	92.0	300.0

In [21]:
```
# Find IDs of positive patients ('SepsisLabel' == 1):
positive_idx = labels.loc[labels['label'] == 1].index
positive = train_df.loc[train_df['PatientID'].isin(positive_idx)]
# Find ID's of negative patients ('SepsisLabel' == 0):
negative_idx = labels.loc[labels['label'] == 0].index
negative = train_df.loc[train_df['PatientID'].isin(negative_idx)]
```

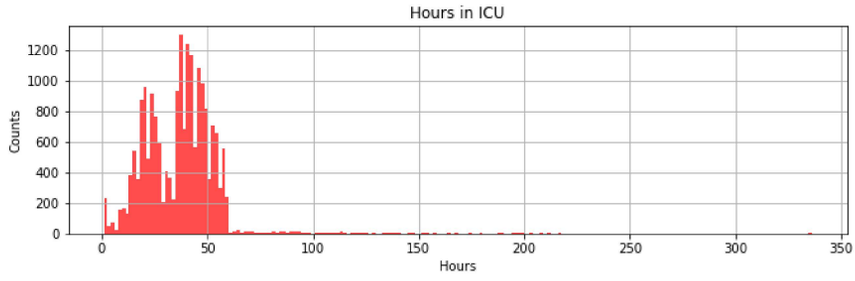
`ICU_duration.loc[k]` returns total hours that `patient_k.psv` was in ICU:

In [25]:
```
ICU_duration = train_df.PatientID.value_counts()
ICU_duration
```

Out[25]:
```
14339    336
12953    336
17925    336
2137     331
151      327
        ...
11242      1
8730       1
12398      1
4568       1
3553       1
Name: PatientID, Length: 20000, dtype: int64
```

In [29]:
```
print("We can observe the long right tail:")
plt.figure(figsize=(11, 3))
ICU_duration.hist(bins='auto', color='Red', alpha=0.7)
plt.title('Hours in ICU')
plt.xlabel('Hours')
plt.ylabel('Counts')
plt.show()
```

```
We can observe the long right tail:
```



For example, patients who left ICU after 1 hour only and their corresponding labels. Note that all the labels appear to be positive:

In [33]:
```
labels.loc[ICU_duration.loc[ICU_duration == 1].index]
```

Out[33]:

	label
2901	1
18393	1
5000	1
6300	1
7324	1
...	...
11242	1
8730	1
12398	1
4568	1
3553	1

198 rows × 1 columns

Approx 200 patients as seen in the histogram above. We can clearly see that there are patients who were in ICU for only one hour, mostly positive (`SepsisLabel == 1`). A possible reason for this is that they passed away, but it's just an assumption.

The average time in ICU:

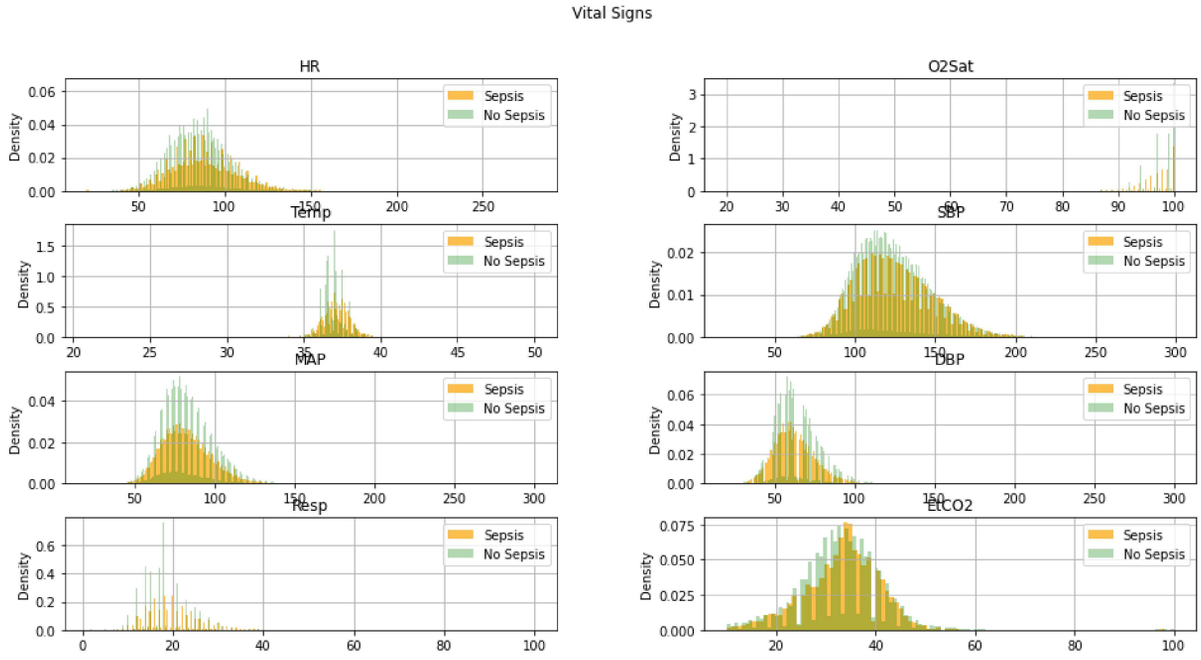
In [34]:
```
print(ICU_duration.mean())
```

```
37.74085
```

In [ ]:
```
df1 = train_df.copy() # df1: train_df with SepsisLabel
df1['SepsisLabel'] = df1['PatientID'].apply(lambda id: int(labels.loc[id]))
```

A deeper look at the distributions of the vital signs:

In [35]:
```
plt.figure(figsize=(16,8))
plt.subplots_adjust(hspace = .3, wspace=.3)
for i, column in enumerate(list(vital), 1):
    plt.subplot(4,2,i)
    positive[column].hist(bins='auto', color='Orange', density=1, alpha=0.7, label='Sepsis')
    negative[column].hist(bins='auto', color='Green', density=1, alpha=0.3, label='No Sepsis')
    plt.legend(loc="upper right")
    plt.title(column)
    plt.ylabel('Density')
plt.suptitle('Vital Signs')
plt.show()
```



Feature selection: We can see some minor changes in the disributuons, depend on the class. for example, `Resp`. We think we won't use `EtCO2` because it's missing as we saw before and the distributions look same. Same routine for `SBP`, the plots look the same. Based on the missing data percents we saw before and the distribution (plots) we decide which features to use. The administrative features doens't tell a lot, so we will drop them too.

Hypothesis Testing:

We will try to compare the means of the heart `HR` rate in the classes. Based on Wikipedia, this feature value should be higher if `SepsisLabel == 1`. We use the last value measured for each patient (to simulate the closest time to start of Sepsis).

In [36]:
```
hr_1 = []
for patient_id, group in positive.groupby('PatientID'):
    if group['HR'].isnull().sum() == len(group.index):
        continue
    valid = group['HR'].last_valid_index()
    hr_1.append(group.iloc[valid]['HR'])
```

In [37]:
```
hr_0 = []
for patient_id, group in negative.groupby('PatientID'):
    if group['HR'].isnull().sum() == len(group.index):
        continue
    valid = group['HR'].last_valid_index()
    hr_0.append(group.iloc[valid]['HR'])
```

Now lets run a simple $T-test$:

In [38]:
```
import scipy.stats as st
print("Confidence intervals:")
print("No Speis:\t{}".format(st.t.interval(0.95, len(hr_0)-1, loc=np.mean(hr_0), scale=st.sem(hr_0))))
print("Sepsis:\t{}".format(st.t.interval(0.95, len(hr_1)-1, loc=np.mean(hr_1), scale=st.sem(hr_1))))
```

```
Confidence intervals:
No Speis:       (82.9474685276275, 83.42186738885081)
Sepsis: (89.20450501643772, 91.30759175775582)
```

In [39]:
```
st.ttest_ind(hr_0, hr_1, equal_var=False, alternative='less')
```

Out[39]:
```
Ttest_indResult(statistic=-12.869248117914394, pvalue=3.677158886752607e-36)
```

We can see how significant the results are. The $Pval$ is in fact 0 and we can say that the mean of the negative class is smaller than the mean of the positive class (based on the last hour, a moment before the first positive label (Sepsis starts). The confidence intervals tells how big is the difference.

TL;DR

Base model

Classification is based on LSTM with some hyperparameter optimization. This is our base model. F_1 score (on dev set) = 0.321 $Recall$ score (on dev set) = 0.312

- We chose RNN because the data is sequential, therefore we assume that each row has an impact on the next row (in other words, context).
- The loss is a binary cross entropy loss, which is particularly good in our case: an unbalanced training set.
- We used resampling (specifically random oversampling) to deal with the imbalance.
- We also measured $Recall$ (besides F_1) because this score proves itself when it comes to medical data and risk of death (by definition, we want to be sure that we didn't miss people who could go at any moment.
- We normalized (mean and std) the data to fit our model better.
- We impute missing data for each patient by similar (e.g. closest) values, a linear interploation backward and forward. In special cases (no cells to copy from) we fill with the mean value of the column.
- As we observed in the analysis, the average time a patient Was in ICU is about 20 hours. In our last run, we used 12 as our num of rows (taken from tail = last hours).
- Regularization: Dropout, $p = 0.75$ (trial and error).
- We didn't use all the features (for more details, please check `Analysis.ipynb`.

Preprocessing:

We work with part of the columns, based on our analysis and further reading. Demographics didn't help to fit our model (actually, the results were better without it) so we dropped those columns. We used some lab-test results that seems to distribute different based on `SepsisLabel` as we saw in plots and stats.

In [1]:
```
from Analysis.utils import concat_data, scale_train_data
from LSTM_Classifier.utils import transform_trim, zip_sequences
from sklearn.model_selection import train_test_split
import numpy as np
import pandas as pd
import importlib
```

In [2]:
```
columns = ["HR", "O2Sat", "Temp", "MAP", "Resp", "AST", "BUN",
           "Alkalinephos", "Calcium", "Creatinine", "Glucose", "Bilirubin_total",
           "Hgb", "PTT", "WBC", "Fibrinogen", "Platelets"]
```

In [3]:
```
df = concat_data(src_dir='Data/sliced/train', dst_dir='LSTM_Data', name='train', impute='lstm')
# df = pd.read_csv('LSTM_Data/train_concatenated.csv', index_col=0)
```

```
100%|██████████| 20000/20000 [01:47<00:00, 186.68it/s]
```

In [4]:
```
tdf = transform_trim(df, columns, r=12)
transformed_df = tdf.copy()
```

```
100%|██████████| 20000/20000 [00:19<00:00, 1005.46it/s]
```

In [5]:
```
transformed_df.head()
```

Out[5]:

	HR	O2Sat	Temp	MAP	Resp	AST	BUN	Alkalinephos	Calcium	Creatinine	Glucose	Bilirubin_total	Hgb	PTT	WBC	Fibrinogen	Platelets	PatientI
0	60.0	94.0	36.110000	74.0	12.0	130.972427	100.0	95.728976	7.9	2.5	78.0	1.426839	9.7	35.901121	11.0	308.205753	158.0	
1	62.0	95.0	36.110000	72.0	12.0	130.972427	100.0	95.728976	7.9	2.5	78.0	1.426839	9.7	35.901121	11.0	308.205753	158.0	
2	63.0	95.0	36.110000	75.0	11.0	130.972427	100.0	95.728976	7.9	2.5	78.0	1.426839	9.7	35.901121	11.0	308.205753	158.0	
3	63.0	97.0	36.110000	81.0	11.0	130.972427	100.0	95.728976	7.9	2.5	78.0	1.426839	9.7	35.901121	11.0	308.205753	158.0	
4	58.0	94.0	36.073333	58.0	10.0	130.972427	100.0	95.728976	7.9	2.5	78.0	1.426839	9.7	35.901121	11.0	308.205753	158.0	

Save feature means (for later use in test):

In [6]:
```
means = transformed_df.mean()
means.to_csv('LSTM_Data/train_means.csv')
```

Normalize the data ($-\mu \quad /\sigma$):

In [7]:
```
# Normalize:
X = scale_train_data(transformed_df, method='standard', path='LSTM_Data')
X['PatientID'] = transformed_df['PatientID']
```

In [8]:
```
X.to_csv('LSTM_Data/train_ready_to_go.csv') # X = pd.read_csv('LSTM_Data/train_ready_to_go.csv', index_col=0)
```

`labels.loc[id]` returns the label of `patient_id.psv`:

In [9]:
```
labels = pd.read_csv('Data/sliced/train_labels.csv', index_col=0)
```

Prepare the data for our sequence model. We'll feed our model with the following tuples (features, label, patient_id). patient_id isn't really necessary for training, except for the last part (exporting labels CSV file) so get things ready now (Later, we'll build our custom Dataset class and use it).

In [10]:
```
sequences = zip_sequences(X, labels, columns)
```

In [11]:
```
train_sequences, val_sequences = train_test_split(sequences, test_size=0.2 , shuffle=True)
print("Training size: {},\tValidation size: {}.".format(len(train_sequences), len(val_sequences)))
```

```
Training size: 16000,   Validation size: 4000.
```

Resampling (oversampling):

Random over-sampling, we sampled 1600 samples from the minor class. We didn't add noise to generate new samples and mitigate overfitting just because the values are sensitive it could do more hurt than good.

In [12]:
```
from sklearn.utils import resample
```

In [13]:
```
positive_idx = [i for i, seq in enumerate(train_sequences) if int(seq[1]) == 1]
positive_sequences = [train_sequences[i] for i in positive_idx]
```

In [14]:
```
negative_idx = [i for i in range(len(train_sequences)) if i not in positive_idx]
negative_sequences = [train_sequences[i] for i in negative_idx]
```

In [15]:
```
resampled_sequences = train_sequences + resample(positive_sequences, n_samples=1600, random_state=42)
```

Dataset:

We implement our own custom DataSet (for PyTorch). Our full implementation can be found in `LSTM_Classifier.SepsisDataset`.

In [16]:
```
from torch.utils.data import DataLoader
from LSTM_Classifier.SepsisDataset import SepsisDataset
```

In [17]:
```
train_dataset = SepsisDataset(train_sequences)
resampled_dataset = SepsisDataset(resampled_sequences)
val_dataset = SepsisDataset(val_sequences)
```

In [18]:
```
BATCH_SIZE = 64
```

In [19]:
```
train_loader = DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE, shuffle=True)
resampled_loader = DataLoader(dataset=resampled_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(dataset=val_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

Model

LSTM of size 100/64 with 4/6 layers and. Dropout regularization of $p = 0.75/0.7$ (found it useful), optimized with Adam optimization algorithm. We took the one who gave the highest score of course. These initial hyper-parameters are nothing more than a guess. Model implementation can be found in `LSTM_Classifier.SequenceModel`. We tried both methods, with resampled data and without.

In [20]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from LSTM_Classifier.SequenceModel import SequenceModel
```

In [21]:
```
NUM_EPOCHS = 20
LEARNING_RATE = 0.001
```

In [22]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [28]:
```
model1 = SequenceModel(n_features=len(columns), n_hidden=100, n_layers=4, dropout=0.75)
model1.to(device)
model2 = SequenceModel(n_features=len(columns), n_hidden=100, n_layers=4, dropout=0.75)
model2.to(device)
model3 = SequenceModel(n_features=len(columns), n_hidden=64, n_layers=6, dropout=0.7)
model3.to(device)
model4 = SequenceModel(n_features=len(columns), n_hidden=64, n_layers=6, dropout=0.7)
model4.to(device)
model5 = SequenceModel(n_features=len(columns), n_hidden=64, n_layers=7, dropout=0.65)
model5.to(device)
models = [model1, model2, model3, model4, model5]
```

Training

Cross entropy loss is known as a useful in case of imbalanced data, that's our case.

In [29]:
```
def train(model, data_loader, criterion = nn.BCEWithLogitsLoss()):
    optimizer = optim.Adam(model.parameters(), lr=LEARNING_RATE)
    f1_scores = []
    recall_scores = []
    for epoch in range(NUM_EPOCHS):
        TP, FP, FN = 0, 0, 0
        for i, (sequences, labels, _) in enumerate(data_loader):
            # Send data to device:
            sequences = sequences.to(device)
            labels = labels.to(device)
            # Set all gradients to zero:
```

```
            optimizer.zero_grad()
            # Calculate loss and backprop:
            outputs = model(sequences)
            loss = criterion(outputs, labels.to(torch.float32))
            loss.backward()
            optimizer.step()
#             # Show progress:
#             if i % 30 == 0:
#                 print ('Epoch: [{}/{}], Step: [{:03d}/{}], Loss: {:.2}'
#                        .format(epoch+1, NUM_EPOCHS, i+1, len(train_dataset)//BATCH_SIZE, loss.item()))
            predictions = torch.where(torch.sigmoid(outputs)>0.5, 1, 0)
            for j in range(labels.shape[0]):
                if labels[j] == 0:
                    if predictions[j] == 1:
                        FP += 1
                else:
                    if predictions[j] == 1:
                        TP += 1
                    else:
                        FN += 1
        print("Counted:\t\tTP={}\tFP={}\tFN={}".format(TP, FP, FN))
        f1_scores.append(TP/(TP+0.5*(FP+FN)))
        recall_scores.append(TP/(TP+FN))
    return f1_scores, recall_scores
```

In [30]:

```
titles = ['($\ell=4, n_h=100, p=0.75$) no resampling',
          '($\ell=4, n_h=100, p=0.75$) oversampling',
          '($\ell=6, n_h=64, p=0.7$) no resampling',
          '($\ell=6, n_h=64, p=0.7$) oversampling',
          '($\ell=7, n_h=64, p=0.65$) oversampling']
loaders = [train_loader, resampled_loader, train_loader, resampled_loader, resampled_loader]
f1 = []
recall = []
```

We can see on the prints below the `TP`, `FP`, `FN` stats every epoch and see how effective is the random oversampling. While the regular train set missed almost every possible prediction, the resampled train predicted right. This is not a fact, we will evaluate later:

In [31]:

```
for i in range(len(models)):
    print('\n\n\n---------------\t'+titles[i]+'\t---------------')
    f1_scores, recall_scores = train(models[i], loaders[i])
    f1.append(f1_scores)
    recall.append(recall_scores)
```

```
--------------- ($\ell=4, n_h=100, p=0.75$) no resampling       ---------------
Counted:        TP=9    FP=47   FN=1140
Counted:        TP=0    FP=0    FN=1149
Counted:        TP=0    FP=1    FN=1149
Counted:        TP=22   FP=10   FN=1127
Counted:        TP=143  FP=43   FN=1006
Counted:        TP=150  FP=25   FN=999
Counted:        TP=141  FP=13   FN=1008
Counted:        TP=170  FP=37   FN=979
Counted:        TP=165  FP=31   FN=984
Counted:        TP=187  FP=32   FN=962
Counted:        TP=225  FP=73   FN=924
Counted:        TP=249  FP=61   FN=900
Counted:        TP=289  FP=113  FN=860
Counted:        TP=302  FP=119  FN=847
Counted:        TP=342  FP=129  FN=807
Counted:        TP=375  FP=131  FN=774
Counted:        TP=400  FP=158  FN=749
Counted:        TP=431  FP=144  FN=718
Counted:        TP=460  FP=148  FN=689
Counted:        TP=483  FP=159  FN=666


--------------- ($\ell=4, n_h=100, p=0.75$) oversampling        ---------------
Counted:        TP=4    FP=8    FN=2745
Counted:        TP=249  FP=190  FN=2500
Counted:        TP=732  FP=318  FN=2017
Counted:        TP=852  FP=390  FN=1897
Counted:        TP=990  FP=436  FN=1759
Counted:        TP=1174 FP=511  FN=1575
Counted:        TP=1270 FP=501  FN=1479
Counted:        TP=1383 FP=539  FN=1366
Counted:        TP=1458 FP=545  FN=1291
Counted:        TP=1580 FP=494  FN=1169
Counted:        TP=1596 FP=465  FN=1153
Counted:        TP=1712 FP=469  FN=1037
Counted:        TP=1805 FP=458  FN=944
Counted:        TP=1865 FP=431  FN=884
Counted:        TP=1929 FP=440  FN=820
Counted:        TP=1964 FP=389  FN=785
Counted:        TP=2059 FP=392  FN=690
Counted:        TP=2070 FP=344  FN=679
Counted:        TP=2085 FP=329  FN=664
Counted:        TP=2122 FP=323  FN=627


--------------- ($\ell=6, n_h=64, p=0.7$) no resampling ---------------
Counted:        TP=3    FP=56   FN=1146
Counted:        TP=0    FP=0    FN=1149
Counted:        TP=0    FP=0    FN=1149
Counted:        TP=69   FP=33   FN=1080
Counted:        TP=138  FP=40   FN=1011
Counted:        TP=133  FP=12   FN=1016
Counted:        TP=146  FP=6    FN=1003
Counted:        TP=146  FP=7    FN=1003
Counted:        TP=145  FP=12   FN=1004
Counted:        TP=153  FP=9    FN=996
Counted:        TP=156  FP=12   FN=993
```

```
Counted:            TP=162  FP=21   FN=987
Counted:            TP=139  FP=7    FN=1010
Counted:            TP=146  FP=11   FN=1003
Counted:            TP=159  FP=18   FN=990
Counted:            TP=164  FP=18   FN=985
Counted:            TP=164  FP=18   FN=985
Counted:            TP=167  FP=20   FN=982
Counted:            TP=201  FP=56   FN=948
Counted:            TP=207  FP=61   FN=942


--------------- ($\ell=6, n_h=64, p=0.7$) oversampling  ---------------
Counted:            TP=18   FP=53   FN=2731
Counted:            TP=90   FP=51   FN=2659
Counted:            TP=404  FP=175  FN=2345
Counted:            TP=544  FP=197  FN=2205
Counted:            TP=600  FP=214  FN=2149
Counted:            TP=687  FP=292  FN=2062
Counted:            TP=873  FP=410  FN=1876
Counted:            TP=1019 FP=516  FN=1730
Counted:            TP=1177 FP=616  FN=1572
Counted:            TP=1323 FP=649  FN=1426
Counted:            TP=1433 FP=688  FN=1316
Counted:            TP=1555 FP=661  FN=1194
Counted:            TP=1627 FP=702  FN=1122
Counted:            TP=1699 FP=681  FN=1050
Counted:            TP=1817 FP=675  FN=932
Counted:            TP=1772 FP=637  FN=977
Counted:            TP=1876 FP=637  FN=873
Counted:            TP=1869 FP=589  FN=880
Counted:            TP=1941 FP=581  FN=808
Counted:            TP=1931 FP=532  FN=818


--------------- ($\ell=7, n_h=64, p=0.65$) oversampling ---------------
Counted:            TP=0    FP=0    FN=2749
Counted:            TP=13   FP=10   FN=2736
Counted:            TP=564  FP=433  FN=2185
Counted:            TP=873  FP=449  FN=1876
Counted:            TP=919  FP=409  FN=1830
Counted:            TP=1074 FP=418  FN=1675
Counted:            TP=1177 FP=453  FN=1572
Counted:            TP=1282 FP=485  FN=1467
Counted:            TP=1328 FP=446  FN=1421
Counted:            TP=1450 FP=432  FN=1299
Counted:            TP=1500 FP=411  FN=1249
Counted:            TP=1567 FP=444  FN=1182
Counted:            TP=1675 FP=438  FN=1074
Counted:            TP=1733 FP=432  FN=1016
Counted:            TP=1772 FP=406  FN=977
Counted:            TP=1840 FP=423  FN=909
Counted:            TP=1842 FP=393  FN=907
Counted:            TP=1948 FP=409  FN=801
Counted:            TP=1986 FP=383  FN=763
Counted:            TP=2017 FP=347  FN=732
```

In [34]:

```
for i in range(len(models)):
    print('---------------------------------')
    print(titles[i]+':\n')
    print("\tF1:\t\t", f1[i][-1])
    print("\tRecall:\t\t", recall[i][-1])
```

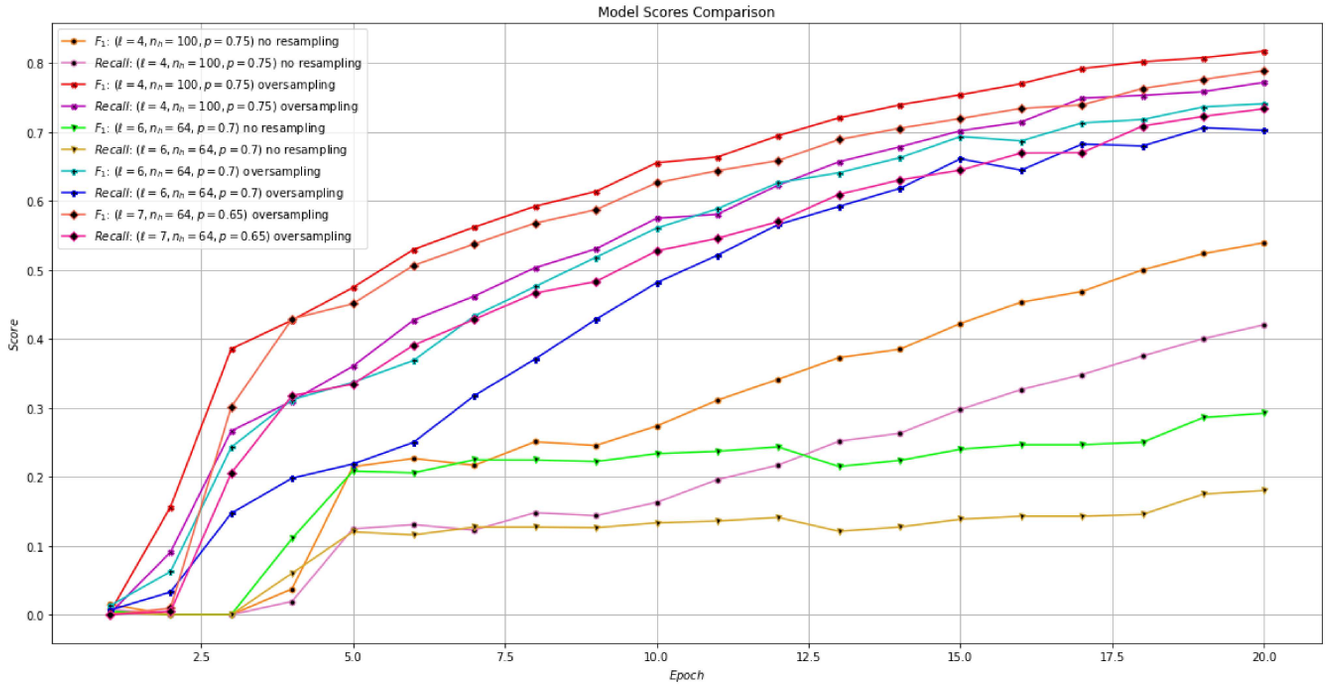
```
---------------------------------
($\ell=4, n_h=100, p=0.75$) no resampling:

        F1:             0.5393634840871022
        Recall:         0.42036553524804177
---------------------------------
($\ell=4, n_h=100, p=0.75$) oversampling:

        F1:             0.817096649980747
        Recall:         0.7719170607493634
---------------------------------
($\ell=6, n_h=64, p=0.7$) no resampling:

        F1:             0.292166549047283
        Recall:         0.1801566579634465
---------------------------------
($\ell=6, n_h=64, p=0.7$) oversampling:

        F1:             0.7409823484267076
        Recall:         0.7024372499090579
---------------------------------
($\ell=7, n_h=64, p=0.65$) oversampling:

        F1:             0.7889692939565812
        Recall:         0.7337213532193525
```

In [35]:

```
import matplotlib.pyplot as plt
```

In [36]:

```
x = range(1,NUM_EPOCHS+1)
_ = plt.figure(figsize=(20, 10))
colors = [['tab:orange', 'tab:pink'], ['r', 'm'], ['lime', 'goldenrod'], ['c', 'b'], ['tomato', 'deeppink']]
markers = ['o', 'X', 'v', 'P', 'D']
for i in range(len(models)):
    _ = plt.plot(x, f1[i], label='$F_1$: '+titles[i], c=colors[i][0], marker=markers[i], ms=5, mfc='k')
    _ = plt.plot(x, recall[i], label='$Recall$: '+titles[i], c=colors[i][1], marker=markers[i], ms=5, mfc='k')
_ = plt.legend()
_ = plt.grid()
_ = plt.xlabel("$Epoch$")
_ = plt.ylabel("$Score$")
_ = plt.title("Model Scores Comparison")
plt.show()
```



Validation

In [37]:
```
import datetime
```

In [38]:
```
def evaluate(model, loader, title):
    model.eval()
    TP, FP, FN = 0, 0, 0
    for i, (sequences, labels, _) in enumerate(loader):
        # Send data to device:
        sequences = sequences.to(device)
        labels = labels.to(device)
        # Predict:
        outputs = model(sequences)
        predictions = torch.FloatTensor([1 if x>0.5 else 0 for x in torch.sigmoid(outputs)])
        labels = labels.reshape(labels.shape[0])
        for j in range(labels.shape[0]):
            if labels[j] == 0:
                if predictions[j] == 1:
                    FP += 1
            else:
                if predictions[j] == 1:
                    TP += 1
                else:
                    FN += 1
    f1 = TP/(TP+0.5*(FP+FN))
    recall = TP/(TP+FN)
    print('---------------\t'+title+'\t---------------')
    print("F1 = {}\nRecall = {}\n".format(f1, recall))
```

In [39]:
```
for i in range (len(models)):
    evaluate(models[i], val_loader, titles[i])
```

```
--------------- ($\ell=4, n_h=100, p=0.75$) no resampling       ---------------
F1 = 0.30288461538461536
Recall = 0.23684210526315788

--------------- ($\ell=4, n_h=100, p=0.75$) oversampling        ---------------
F1 = 0.32170542635658916
Recall = 0.31203007518796994

--------------- ($\ell=6, n_h=64, p=0.7$) no resampling ---------------
F1 = 0.19047619047619047
Recall = 0.10526315789473684

--------------- ($\ell=6, n_h=64, p=0.7$) oversampling  ---------------
F1 = 0.31956912028725315
Recall = 0.33458646616541354

--------------- ($\ell=7, n_h=64, p=0.65$) oversampling ---------------
F1 = 0.27944111776447106
Recall = 0.2631578947368421
```

As we can see, the oversampling plays key role here in the results. We don't optimize the parameters anymore because this is our base model, specific with size 100 (hidden), 4 layers, dropout w.p. 0.75 and random oversampling which gave us the score $F_1 = 0.321$ on the validation set. That's it. Next we'll try different models from a different approach (ignoring the fact that the data is sequential and preprocess in a new way based on some important stats).

In [40]:
```
basename = "LSTM_Data/model_created"
suffix = datetime.datetime.now().strftime("%H_%M")
filename = "_".join([basename, suffix])

torch.save(model2.state_dict(), filename+'.pkl')
```

TL;DR (Final model):

This is our final model. Classification is based on Random Forest.

- We tried some different classifiers (MLP, LR, XGBOOST) and we found that the RF gave us the best F_1 score.
- Random forest is a simple model, yet powerful and easy to train, and can handle with imbalanced data.
- We handle with the imbalance by some resampling methods (and choose the best).
- We fine-tune our hyper-parameters with Randomized search.
- We didn't use all the features (only part of the vital signs and laboratory values. For more details, plese check our first analysis.
- We measured 3 diffrenet metrics: F_1 (requested), $Recall$, as explained before (base model section), this is a significant score when dealing with critical classification of medical data. $BalancedAccuracy$ - kind of accuracy, but computed for imbalanced data (like ours). We know that regular accuracy is useless here and non-informative, so this one more robust to the imbalance (but still not standalone).
- We split our training set to train:validation with ratio of 80:20.
- We transform our data in a a special way: for every patient, we took only the main stats (like mean, std, median, max, min 0.25 qunatile and 0.75 quantile) of each column (18 columns) and now we have 18*7=126 features for every patient. Final result contains 1 row of these stats for every patient (instead of full dataframe). This made the data more simple to use, gave us the oppurunity to resample the data with `imbalanced-learn` package and finally gave us better result than using the data as is.

Preprocessing:

In [1]:
```
import pandas as pd
import numpy as np
import joblib
from Analysis.utils import concat_data
from RF_Classifier.utils import transform_stats
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

The columns we chose to work with, based on our first analysis and some trials. We work with all the vital signs and with the lab-test result:

In [2]:
```
columns = ["HR", "O2Sat", "Temp", "MAP", "Resp", "AST", "BUN",
           "Alkalinephos", "Calcium", "Creatinine", "Glucose", "Bilirubin_total",
           "Hgb", "PTT", "WBC", "Fibrinogen", "Platelets", "ICULOS"]
```

Load data and impute missing data (read full docs of 'concat_data' in our `RF_Classifier.utils` file, simply fill backward&forward / interpolation of nearest values according to the state of the column):

In [3]:
```
df = concat_data(src_dir='Data/sliced/train', dst_dir='RF_Data', name='train', impute='rf')
```

```
100%|████████████████████████████████████████████████████████| 20000/20000 [09:09<00:00, 36.39it/s]
```

Transform data to stats only, each patient has now mean, median, std, etc. summerize all his hours in ICU (read docs of 'transform_stats' in our `RF_Classifier.utils` file).

In [5]:
```
tdf = transform_stats(df, columns)
transformed_df = tdf.copy()
```

```
100%|████████████████████████████████████████████████████████| 20000/20000 [15:06<00:00, 22.06it/s]
```

In [47]:
```
transformed_df.head(2)
```

Out[47]:

	HR_count	HR_mean	HR_std	HR_min	HR_25%	HR_50%	HR_75%	HR_max	O2Sat_count	O2Sat_mean	...	ICULOS_count	ICULOS_mean	ICULOS_std	ICULOS_min
0	23.0	60.956522	7.957148	54.0	57.0	60.0	62.5	94.0	23.0	97.086957	...	23.0	12.0	6.782330	1.0
1	25.0	87.760000	6.808573	79.0	82.0	85.0	93.0	101.0	25.0	100.000000	...	25.0	14.0	7.359801	2.0

2 rows × 146 columns

Load labels and sort corresponding to the samples (`train_labels.csv` from our analysis):

In [7]:
```
# train_labels.loc[id: int] returns the label of patient_id.psv
train_labels = pd.read_csv('Data/sliced/train_labels.csv', index_col=0)
```

In [8]:
```
# Add labels
transformed_df['SepsisLabel'] = transformed_df['PatientID'].apply(lambda k: int(train_labels.loc[k]))
```

In [9]:
```
transformed_df.to_csv('RF_Data/train_ready_to_go.csv')
```

In [10]:
```
X = np.array(transformed_df.drop(columns=['PatientID', 'SepsisLabel']))
y = np.array(transformed_df['SepsisLabel'])
```

In [62]:
```
# Split:
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, shuffle=True)
print("Training size: {},\tValidation size: {}.".format(X_train.shape[0], X_val.shape[0]))
```

```
Training size: 16000,   Validation size: 4000.
```

Resampling:

We will try some methods both over-sampling and combination of under-over sampling. Some techniques used below are heuristic. We found this library `imbalanced-learn` (MIT-licensed) very useful, designed to fit sklearn and has great documentation with plots and math formulation. Of course, we'll choose the

method that will give the highest score on the validation set after training.

In [12]:
```
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN
from imblearn.combine import SMOTETomek, SMOTEENN
```

In [63]:
```
X1, y1 = SMOTE(random_state=42).fit_resample(X_train, y_train)
X2, y2 = ADASYN(random_state=42).fit_resample(X_train, y_train)
ros = RandomOverSampler(random_state=42) # randomly over-sampling with replacement:
X3, y3 = ros.fit_resample(X_train, y_train)
smote_tomek = SMOTETomek(random_state=42) # Combined method: synthetic minority over-sampling technique:
X4, y4 = smote_tomek.fit_resample(X_train, y_train)
smote_enn = SMOTEENN(random_state=42) # Combined method: adaptive synthetic sampling:
X5, y5 = smote_enn.fit_resample(X_train, y_train)
```

Training:

3 important metrics:

- F_1: the main metric of the assignment.
- $Recall$: as explained before (base model section), this is a significant score when dealing with critical classification of medical data.
- $Accuracy_{balanced}$: kind of accuracy, but computed for imbalanced data (like ours). We know that regular accuracy is useless here and non-informative, so this one more robust to the imbalance (but still not standalone).

Random forest:

In [17]:
```
from sklearn.metrics import f1_score, balanced_accuracy_score, recall_score
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
```

First, we will tune our random forest (find best hyper-params) with our original train set, witout resampling. Then, we will take these hyper-params and try all the resampling techniques above. Find best hyper parameters:

In [18]:
```
n_estimators = [int(x) for x in np.linspace(start=100, stop=700, num=5)] # Number of trees in random forest
max_depth = [int(x) for x in np.linspace(start=10, stop=55, num=5)] # Max number of levels in tree
max_depth.append(None)
min_samples_split = [2, 3, 4, 5] # Min numbers of samples required to split a node
min_samples_leaf = [1, 4, 8] # Min numbers of sample required at each leaf node
```

In [19]:
```
random_grid = {'n_estimators': n_estimators, 'max_depth': max_depth,
               'min_samples_split': min_samples_split, 'min_samples_leaf': min_samples_leaf}
```

In [20]:
```
RFC = RandomForestClassifier()
RFC_grid = RandomizedSearchCV(estimator=RFC, param_distributions=random_grid,
                              n_iter=10, random_state=42, n_jobs=-1, scoring='f1', error_score='raise')
```

In [ ]:
```
RFC_grid.fit(X_train, y_train)
```

In [23]:
```
# Base model:
base_model = RandomForestClassifier()
base_model.fit(X_train, y_train)
f1_base = f1_score(y_val, base_model.predict(X_val))
# Best random
best_random = RFC_grid.best_estimator_
f1_best = f1_score(y_val, best_random.predict(X_val))
print("F1 base:\t{}\t\tF1 best:\t{}".format(f1_base, f1_best))
```

```
F1 base:        0.6804597701149425              F1 best:        0.6758620689655173
```

Now, lets use these hyper-parameters to find the best resampling method:

In [64]:
```
RFC0 = RandomForestClassifier(n_estimators=400 ,n_jobs=-1)
RFC1 = RandomForestClassifier(n_estimators=400 ,n_jobs=-1)
RFC2 = RandomForestClassifier(n_estimators=400 ,n_jobs=-1)
RFC3 = RandomForestClassifier(n_estimators=400 ,n_jobs=-1)
RFC4 = RandomForestClassifier(n_estimators=400 ,n_jobs=-1)
RFC5 = RandomForestClassifier(n_estimators=400 ,n_jobs=-1)
```

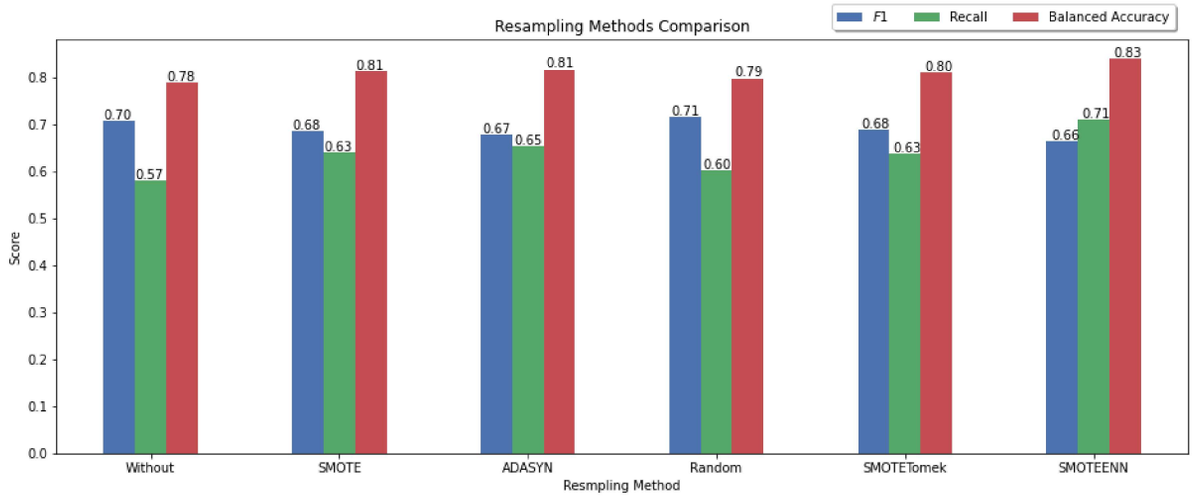
In [65]:
```
RFC = [RFC0, RFC1, RFC2, RFC3, RFC4, RFC5]
data = [(X_train, y_train), (X1, y1), (X2, y2), (X3, y3), (X4, y4), (X5, y5)]
text = ['Without','SMOTE', 'ADASYN', 'Random', 'SMOTETomek', 'SMOTEENN']
```

Train:

In [66]:
```
scores = {'$F1$': [], 'Recall': [], 'Balanced Accuracy': []}
for i in range(len(RFC)):
    X_train = data[i][0]
    labels = data[i][1]
    RFC[i].fit(X_train, labels)
    train_pred = RFC[i].predict(X_train)
    val_pred = RFC[i].predict(X_val)
    print('----- '+text[i]+' resampling: -----')
    score1 = f1_score(y_val, val_pred)
    print('\tTrain F1 score:', f1_score(labels, train_pred), ' \t\tValidation F1 score:', score1)
    scores['$F1$'].append(score1)
    score2 = balanced_accuracy_score(y_val, val_pred)
    print('\tTrain balanced-accuracy:', balanced_accuracy_score(labels, train_pred), ' \tValidation balanced-accuracy:', score2)
    scores['Balanced Accuracy'].append(score2)
    score3 = recall_score(y_val, val_pred)
    print('\tTrain Recall:', recall_score(labels, train_pred), ' \t\tValidation Recall:', score3)
    scores['Recall'].append(score3)
```

```
----- Without resampling: -----
        Train F1 score: 1.0             Validation F1 score: 0.7068965517241379
        Train balanced-accuracy: 1.0    Validation balanced-accuracy: 0.7874658597542947
        Train Recall: 1.0               Validation Recall: 0.5795053003533569
----- SMOTE resampling: -----
        Train F1 score: 1.0             Validation F1 score: 0.6856060606060606
        Train balanced-accuracy: 1.0    Validation balanced-accuracy: 0.8111788925108683
        Train Recall: 1.0               Validation Recall: 0.6395759717314488
----- ADASYN resampling: -----
        Train F1 score: 1.0             Validation F1 score: 0.676416819012797
        Train balanced-accuracy: 1.0    Validation balanced-accuracy: 0.8162282740650112
        Train Recall: 1.0               Validation Recall: 0.6537102473498233
----- Random resampling: -----
        Train F1 score: 1.0             Validation F1 score: 0.7142857142857143
        Train balanced-accuracy: 1.0    Validation balanced-accuracy: 0.797259463966058
        Train Recall: 1.0               Validation Recall: 0.6007067137809188
----- SMOTETomek resampling: -----
        Train F1 score: 1.0             Validation F1 score: 0.6883365200764817
        Train balanced-accuracy: 1.0    Validation balanced-accuracy: 0.80995017639325
        Train Recall: 1.0               Validation Recall: 0.6360424028268551
----- SMOTEENN resampling: -----
        Train F1 score: 1.0             Validation F1 score: 0.6644628099173555
        Train balanced-accuracy: 1.0    Validation balanced-accuracy: 0.838847107787636
        Train Recall: 1.0               Validation Recall: 0.7102473498233216
```

In [67]:
```
import matplotlib.pyplot as plt
plt.style.use('seaborn-deep')
df = pd.DataFrame(scores, index=text)
ax = df.plot(kind='bar', stacked=False, figsize=(16, 6), rot=0, xlabel='Resmpling Method', ylabel='Score',
             title='Resampling Methods Comparison')
for p in ax.patches:
    ax.annotate(str(p.get_height())[:4], (p.get_x() * 1.005, p.get_height() * 1.005))
ax.legend(loc='upper left', bbox_to_anchor=(0.68, 1.1),
          ncol=3, fancybox=True, shadow=True)
plt.show()
```



We can clearly observe that the first model (without resampling) gave us $F_1 = 0.7068$ which is a bit higher compare to the other methods. The model that achieved the highest score is trained with random sampling and achieved $F_1 = 0.7142$ It's interesting that the random is better than math based resampling techniques, but makes sense. Saving the best model (random oversampling):

In [68]:
```
[joblib.dump(RFC[i], 'RF_Data/'+text[i]+'.pkl') for i in range(len(RFC))]
print("")
```

Conclusion

1. We conclude that when we have imbalanced data (datasets where the target class has an uneven distribution of observations, i.e one class label has a very high number of observations and the other has a very low number of observations like our dataset), we can't deal with it in the original way (take every patient dateframe and labels and fit in with the models). It gave us a low value of $F1$ score (0.003). So because of that we minimize each patient's data into a single line that has its own data statistics (include those that summarize the central tendency).
2. On our base model (LSTM) we tried to insert the data to the model as is was (data frame for each patient). This model gave us a bad $F1$ (0.32) . We concluded that it is not customary to use this data set in this way. We doubled our $F1$ score from our base model to our final try.
3. We concluded that data preprocessing and removal of irrelevant data (does not improve or interfere with training) can improve the training process, that is why we took the data that have different distribution between Sepsis patients and non-Sepsis patients.
4. Resampling didn't contribute in our final model, but did in our base model (in the LSTM attempts, the random oversampling made all the diffrence).
5. Meidcal data is interesting, mainly because it deals with health of humans which is important and it also challenging because the missing data and the inaccuracy of it. In this field, the parameter of interest (Sepsis in our case) is almost always the minor class, and this imbalance doesn't good to us, the researchers.