Project 3.2

Derek Caramella

DSCC 275: Times Series Analysis & Forecasting

Due: 12 December 2021

Overview

In this project, you will work with LSTM-based autoencoders to classify human heart beats for heart disease diagnosis. The dataset contains 5,000 Time Series examples with 140 timesteps. Each time-series is an ECG or EKG signal that corresponds to a single heartbeat from a single patient with congestive heart failure. An electrocardiogram (ECG or EKG) is a test that checks how your heart is functioning by measuring the electrical activity of the heart. With each heart beat, an electrical impulse (or wave) travels through your heart. This wave causes the muscle to squeeze and pump blood from the heart. There are 5 types of hearbeats (classes) that can be classified: i) Normal (N); ii) R-on-T Premature Ventricular Contraction (R on-T PVC); iii) Premature Ventricular Contraction (PVC); iv) Supra-ventricular Premature or Ectopic Beat (SP or EB); v) Unclassified Beat (UB). The shape of the time-series and the position of the impulses allows doctors to diagnose these different conditions. For the purposes of this project, we are interested in 2 classes: *Normal* and *Abnormal* (which includes class 2-5 above merged).

This is an example of an anomaly detection problem where class imbalance exists, i.e. number of each of the individual positive (abnormal) instances are smaller than the normal case. The autoencoder approach is suited well for such **applications of anomaly detection**. In anomaly detection, we learn the pattern of a normal process. Anything that does not follow this pattern is classified as an anomaly. For a binary classification of rare events, we can use a similar approach using autoencoders.

A sample code example (in Python) implementation of auto-encoder “AutoEncoders_anomaly_detection_ecg_SAMPLE.py” is provided. Review and run the code and answer the following questions:

In [2]:

```
import torch
import copy
import numpy as np
import pandas as pd
import seaborn as sns
from pylab import rcParams
import matplotlib.pyplot as plt
from matplotlib import rc
from sklearn.model_selection import train_test_split
from torch import nn, optim
import torch.nn.functional as F
from arff2pandas import a2p
import warnings
import pickle


warnings.filterwarnings('ignore')  # Surpress warnings
plt.rcParams['font.family'] = 'Times New Roman'  # Set plt shows font to Times New Roma
```

In [2]:

```
sns.set(style='whitegrid', palette='muted', font_scale=1.2, font='Times New Roman')
HAPPY_COLORS_PALETTE = ["#01BEFE", "#FFDD00", "#FF7D00", "#FF006D", "#ADFF02", "#8F00FF
sns.set_palette(sns.color_palette(HAPPY_COLORS_PALETTE))
rcParams['figure.figsize'] = 12, 8

RANDOM_SEED = 42
np.random.seed(RANDOM_SEED)
torch.manual_seed(RANDOM_SEED)
```

Out[2]:

```
<torch._C.Generator at 0x2c5aaa2d230>
```

In [3]:

```
#Load the arff files into Pandas data frames / Change Path as needed
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
with open(r'ECG5000_TRAIN.arff') as f:
    train = a2p.load(f)

with open(r'ECG5000_TEST.arff') as f:
    test = a2p.load(f)
```

In [4]:

```
# We'll combine the training and test data into a single data frame.
# This will give us more data to train our Autoencoder. We'll also shuffle it

df = train.append(test)
df = df.sample(frac=1.0)
df.shape

df.head()
```

Out[4]:

	att1@NUMERIC	att2@NUMERIC	att3@NUMERIC	att4@NUMERIC	att5@NUMERIC	att6@NUMER
1001	1.469756	-1.048520	-3.394356	-4.254399	-4.162834	-3.8225
2086	-1.998602	-3.770552	-4.267091	-4.256133	-3.515288	-2.5545
2153	-1.187772	-3.365038	-3.695653	-4.094781	-3.992549	-3.4253
555	0.604969	-1.671363	-3.236131	-3.966465	-4.067820	-3.5518
205	-1.197203	-3.270123	-3.778723	-3.977574	-3.405060	-2.3926

5 rows × 141 columns

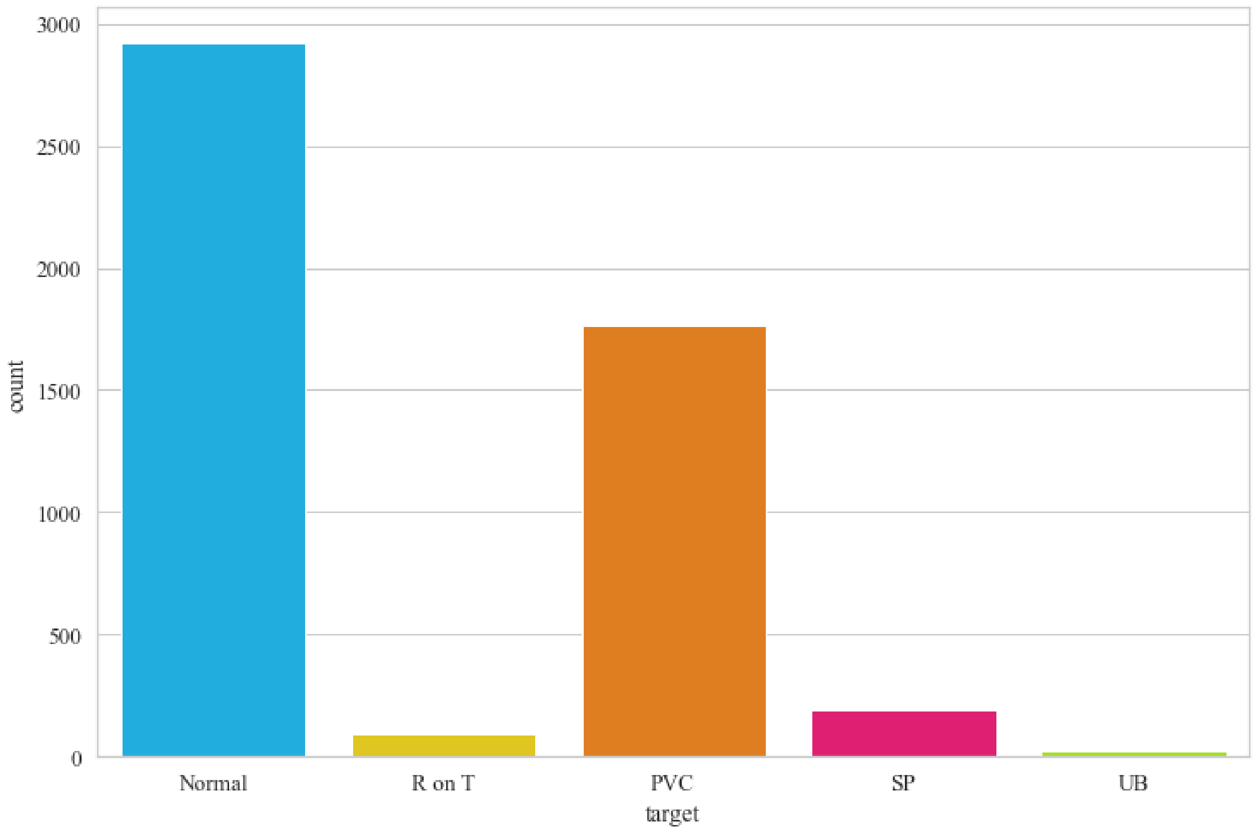
In [5]:

```
# We have 5,000 examples. Each row represents a single heartbeat record. Let's name the

CLASS_NORMAL = 1
class_names = ['Normal','R on T','PVC','SP','UB']
```

In [6]:
```
# Next, we'll rename the last column to `target`, so its easier to reference it

new_columns = list(df.columns)
new_columns[-1] = 'target'
df.columns = new_columns
```

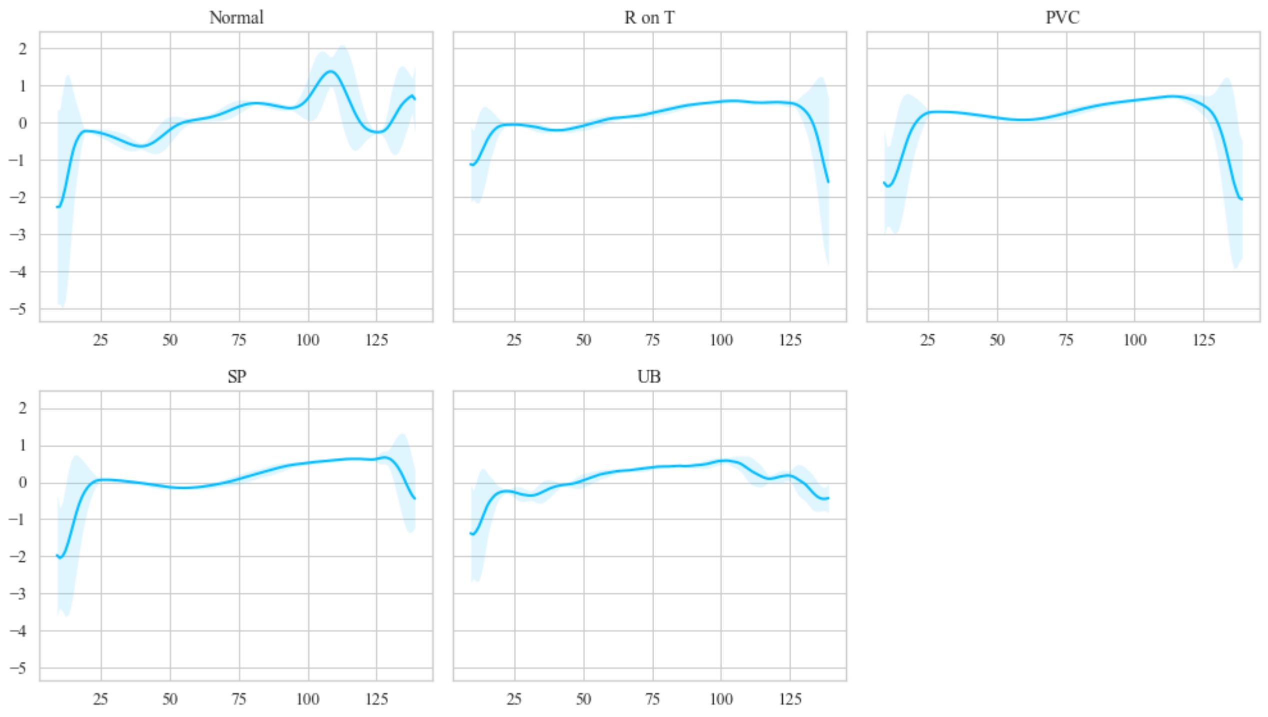
Exploratory Data Analysis

In [7]:
```
# Check how many examples for each heartbeat class do we have:
df.target.value_counts()

# plot the results
ax = sns.countplot(x=df.target)
ax.set_xticklabels(class_names)
plt.show()
```



The normal class, has by far, the most examples.

In [8]:
```
# Let's have a look at an averaged (smoothed out with one standard deviation on top and
# Time Series for each class:

def plot_time_series_class(data, class_name, ax, n_steps=10):
    time_series_df = pd.DataFrame(data)

    smooth_path = time_series_df.rolling(n_steps).mean()
    path_deviation = 2 * time_series_df.rolling(n_steps).std()

    under_line = (smooth_path - path_deviation)[0]
    over_line = (smooth_path + path_deviation)[0]
    ax.plot(smooth_path, linewidth=2)
    ax.fill_between(
        path_deviation.index,
        under_line,
        over_line,
        alpha=.125)
    ax.set_title(class_name)


classes = df.target.unique()

fig, axs = plt.subplots(
    nrows=len(classes) // 3 + 1,
    ncols=3,
    sharey=True,
    figsize=(14, 8))

for i, cls in enumerate(classes):
    ax = axs.flat[i]
    data = df[df.target == cls].drop(labels='target', axis=1).mean(axis=0).to_numpy()
    plot_time_series_class(data, class_names[i], ax)

fig.delaxes(axs.flat[-1])
fig.tight_layout();
plt.show()
```



Data Preprocessing

In [9]:
```
# Let's get all normal heartbeats and drop the target (class) column:
normal_df = df[df.target == str(CLASS_NORMAL)].drop(labels='target', axis=1)
normal_df.shape
```

Out[9]:
```
(2919, 140)
```

In [10]:
```
# Merge all other classes and mark them as anomalies
anomaly_df = df[df.target != str(CLASS_NORMAL)].drop(labels='target', axis=1)
anomaly_df.shape
```

Out[10]:
```
(2081, 140)
```

In [11]:
```

# Split the normal examples into train, validation and test sets
train_df, val_df = train_test_split(
    normal_df,
    test_size=0.15,
    random_state=RANDOM_SEED)

val_df, test_df = train_test_split(
    val_df,
    test_size=0.33,
    random_state=RANDOM_SEED)
```

In [12]:
```

def create_dataset(df):
    """Convert our examples into tensors, so we can use them to train our Autoencoder."
    sequences = df.astype(np.float32).to_numpy().tolist()
    dataset = [torch.tensor(s).unsqueeze(1).float() for s in sequences]
    n_seq, seq_len, n_features = torch.stack(dataset).shape
    return dataset, seq_len, n_features
```

In [13]:
```

# Each Time Series converted to a 2D Tensor in the shape *sequence length* x *number of

# Create Train, Val and Test datasets:
train_dataset, seq_len, n_features = create_dataset(train_df)
val_dataset, _, _ = create_dataset(val_df)
test_normal_dataset, _, _ = create_dataset(test_df)
test_anomaly_dataset, _, _ = create_dataset(anomaly_df)
```

LSTM Autoencoder

In [14]:
```

class Encoder(nn.Module):
    """The general Autoencoder architecture consists of two components.
    An *Encoder* that compresses the input and a *Decoder* that tries to reconstruct it
    The *Encoder* uses LSTM layers to compress the Time Series data input."""
    def __init__(self, seq_len, n_features, embedding_dim=64):
        super(Encoder, self).__init__()

        self.seq_len, self.n_features = seq_len, n_features
        # self.embedding_dim, self.hidden_dim = embedding_dim, 2 * embedding_dim
        self.embedding_dim, self.hidden_dim = embedding_dim, embedding_dim

        self.rnn1 = nn.LSTM(
            input_size=n_features,
            hidden_size=self.hidden_dim,
            num_layers=1,
            batch_first=True)

    def forward(self, x):
        x = x.reshape((1, self.seq_len, self.n_features))
```

```
#         x, (_, _) = self.rnn1(x)
        x, (hidden_n, _) = self.rnn1(x)
#         x, (hidden_n, _) = self.rnn2(x)

        return hidden_n.reshape((self.n_features, self.embedding_dim))
```

In [15]:

```
class Decoder(nn.Module):
    """Decode the compressed representation using a *Decoder*.
    Our Decoder contains LSTM layer and an output layer that gives the final reconstruc
    def __init__(self, seq_len, input_dim=64, n_features=1):
        super(Decoder, self).__init__()

        self.seq_len, self.input_dim = seq_len, input_dim
#         self.hidden_dim, self.n_features = 2 * input_dim, n_features
        self.hidden_dim, self.n_features = input_dim, n_features

        self.rnn1 = nn.LSTM(
            input_size=input_dim,
            hidden_size=input_dim,
            num_layers=1,
            batch_first=True)

#         self.rnn2 = nn.LSTM(
#             input_size=input_dim,
#             hidden_size=self.hidden_dim,
#             num_layers=1,
#             batch_first=True)

        self.output_layer = nn.Linear(self.hidden_dim, n_features)

    def forward(self, x):
        x = x.repeat(self.seq_len, self.n_features)
        x = x.reshape((self.n_features, self.seq_len, self.input_dim))

        x, (hidden_n, cell_n) = self.rnn1(x)
#         x, (hidden_n, cell_n) = self.rnn2(x)
        x = x.reshape((self.seq_len, self.hidden_dim))

        return self.output_layer(x)
```

In [16]:

```
class RecurrentAutoencoder(nn.Module):
    """Wrap everything into an easy to use module"""
    def __init__(self, seq_len, n_features, embedding_dim=64):
        super(RecurrentAutoencoder, self).__init__()

        self.encoder = Encoder(seq_len, n_features, embedding_dim).to(device)
        self.decoder = Decoder(seq_len, embedding_dim, n_features).to(device)

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)

        return x
```

Training

In [17]:
```
def train_model(model, train_dataset, val_dataset, n_epochs):
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    criterion = nn.L1Loss(reduction='sum').to(device)
    history = dict(train=[], val=[])

    best_model_wts = copy.deepcopy(model.state_dict())
    best_loss = 10000.0

    for epoch in range(1, n_epochs + 1):
        model = model.train()

        train_losses = []

        for seq_true in train_dataset:
            optimizer.zero_grad()

            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            loss.backward()
            optimizer.step()

            train_losses.append(loss.item())

        val_losses = []
        model = model.eval()
        with torch.no_grad():
            for seq_true in val_dataset:

                seq_true = seq_true.to(device)
                seq_pred = model(seq_true)

                loss = criterion(seq_pred, seq_true)
                val_losses.append(loss.item())

        train_loss = np.mean(train_losses)
        val_loss = np.mean(val_losses)

        history['train'].append(train_loss)
        history['val'].append(val_loss)

        if val_loss < best_loss:
            best_loss = val_loss
            best_model_wts = copy.deepcopy(model.state_dict())

        print(f'Epoch {epoch}: train loss {train_loss} val loss {val_loss}')

    model.load_state_dict(best_model_wts)
    return model.eval(), history
```

Question 2

In the above example, the embedding dimension (i.e. output length of encoder and input length of decoder) was set constant at 8.

a) Embedding dimension length is typically an important hyperparameter that can affect the performance of the technique. Vary the embedding dimension from 2 to 8 in increments of 2 and report the training and validation loss after 25 epochs.

In [18]:

```
# Reporting lists

embedding_dimension_2_train_loss = []
embedding_dimension_2_val_loss = []
embedding_dimension_2_normal_recall = 0
embedding_dimension_2_anomaly_recall = 0

embedding_dimension_4_train_loss = []
embedding_dimension_4_val_loss = []
embedding_dimension_4_normal_recall = 0
embedding_dimension_4_anomaly_recall = 0

embedding_dimension_6_train_loss = []
embedding_dimension_6_val_loss = []
embedding_dimension_6_normal_recall = 0
embedding_dimension_6_anomaly_recall = 0

embedding_dimension_8_train_loss = []
embedding_dimension_8_val_loss = []
embedding_dimension_8_normal_recall = 0
embedding_dimension_8_anomaly_recall = 0
```

Embedding Dimension 2

In [19]:

```
# Our Autoencoder passes the input through the Encoder and Decoder. Let's create an ins

model = RecurrentAutoencoder(seq_len, n_features, 2)
model = model.to(device)
```

At each epoch, the training process feeds our model with all training examples and evaluates the performance on the validation set. Note that we're using a batch size of 1 (our model sees only 1 sequence at a time). We also record the training and validation set losses during the process.

Note that we're minimizing the L1Loss, which measures the MAE (mean absolute error). Why? The reconstructions seem to be better than with MSE (mean squared error).

We'll get the version of the model with the smallest validation error. Let's do some training:

In [20]:

```
model, history = train_model(
    model,
    train_dataset,
    val_dataset,
    n_epochs=25)
```

```
embedding_dimension_2_train_loss = history['train'].copy()
embedding_dimension_2_val_loss = history['val'].copy()
```

```
Epoch 1: train loss 93.41885137039056 val loss 86.77800396603529
Epoch 2: train loss 72.17759312818436 val loss 69.29357194086798
Epoch 3: train loss 69.25752414056055 val loss 68.8441378752933
Epoch 4: train loss 68.25562451665125 val loss 68.883027385933
Epoch 5: train loss 67.91505854190333 val loss 69.27835809492821
Epoch 6: train loss 67.59339387548494 val loss 68.9324425056119
Epoch 7: train loss 67.40365366876149 val loss 69.01957558120884
Epoch 8: train loss 67.22636068852282 val loss 69.28097323264686
Epoch 9: train loss 67.07850279648522 val loss 68.96694644237948
Epoch 10: train loss 66.96674830795344 val loss 67.58654556014025
Epoch 11: train loss 66.8991280201701 val loss 66.9944656684537
Epoch 12: train loss 66.83337108822707 val loss 67.7595143887777
Epoch 13: train loss 66.71373047342419 val loss 67.45298815261789
Epoch 14: train loss 66.60439007938605 val loss 65.88095761972889
Epoch 15: train loss 66.50556692871439 val loss 65.80937776793392
Epoch 16: train loss 66.40613906626642 val loss 65.82687030310517
Epoch 17: train loss 66.33911164025058 val loss 65.6951421666064
Epoch 18: train loss 66.22237910863038 val loss 65.63371737743807
Epoch 19: train loss 66.13086608800616 val loss 65.55129364326139
Epoch 20: train loss 66.02187077262045 val loss 65.45048502118108
Epoch 21: train loss 65.92897178991241 val loss 65.36711675233808
Epoch 22: train loss 65.78494029139281 val loss 65.54207505626484
Epoch 23: train loss 65.68827678214538 val loss 65.30189098839874
Epoch 24: train loss 65.70056736771977 val loss 65.65064792828348
Epoch 25: train loss 65.65717878576153 val loss 65.27996522818816
```

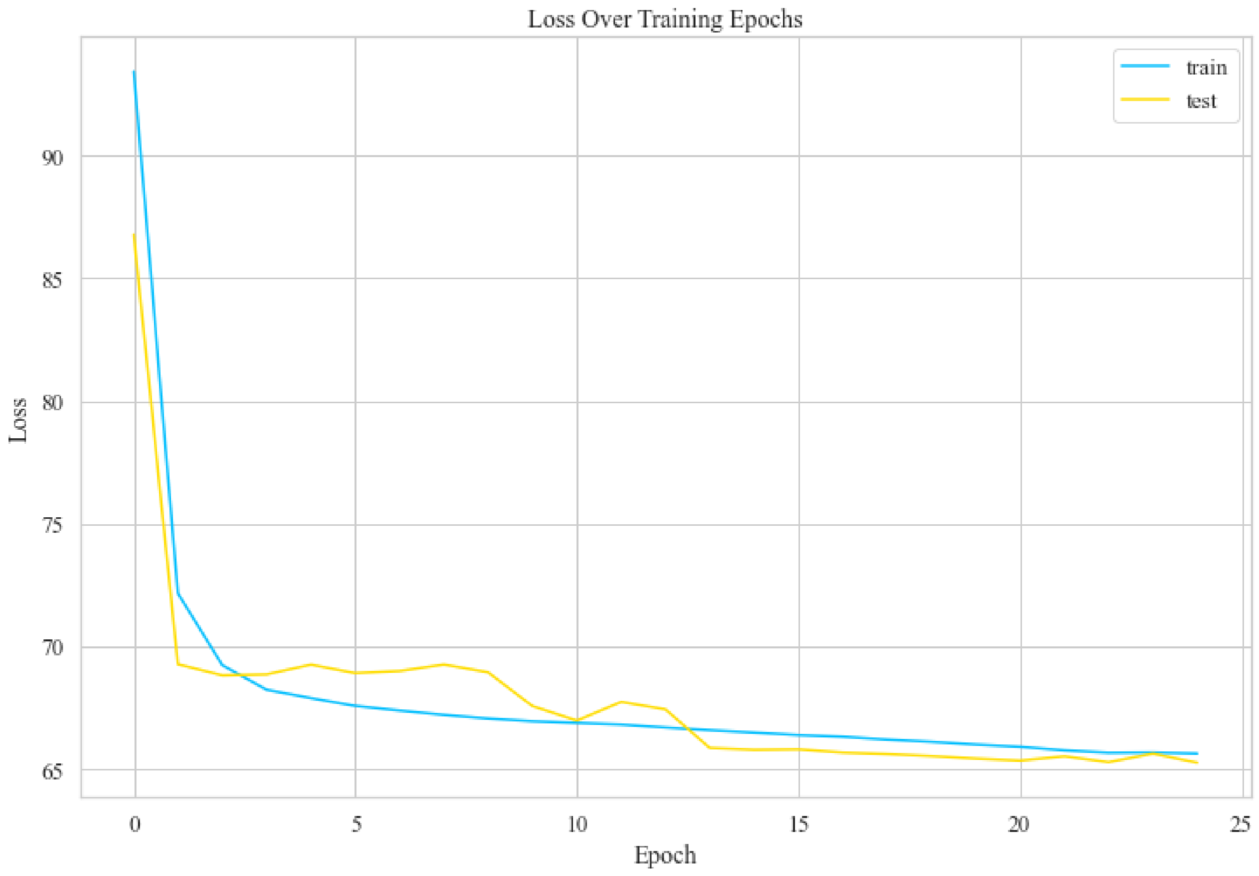
Choosing a Threshold

In [21]:

```
def predict(model, dataset):
    """The reconstruction error on the training set. Let's start by writing a helper fu
    predictions from our model"""
    predictions, losses = [], []
    criterion = nn.L1Loss(reduction='sum').to(device)
    with torch.no_grad():
        model = model.eval()
        for seq_true in dataset:
            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            predictions.append(seq_pred.cpu().numpy().flatten())
            losses.append(loss.item())
    return predictions, losses
```

In [22]:

```
ax = plt.figure().gca()

ax.plot(history['train'])
ax.plot(history['val'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'])
plt.title('Loss Over Training Epochs')

plt.show()
```



Save the Model

In [23]:
```
MODEL_PATH = 'model25_2.pth'
torch.save(model, MODEL_PATH)
```

Load the Model

In [24]:
```
MODEL_PATH = 'model25_2.pth'
model = torch.load(MODEL_PATH)
```

Evaluation

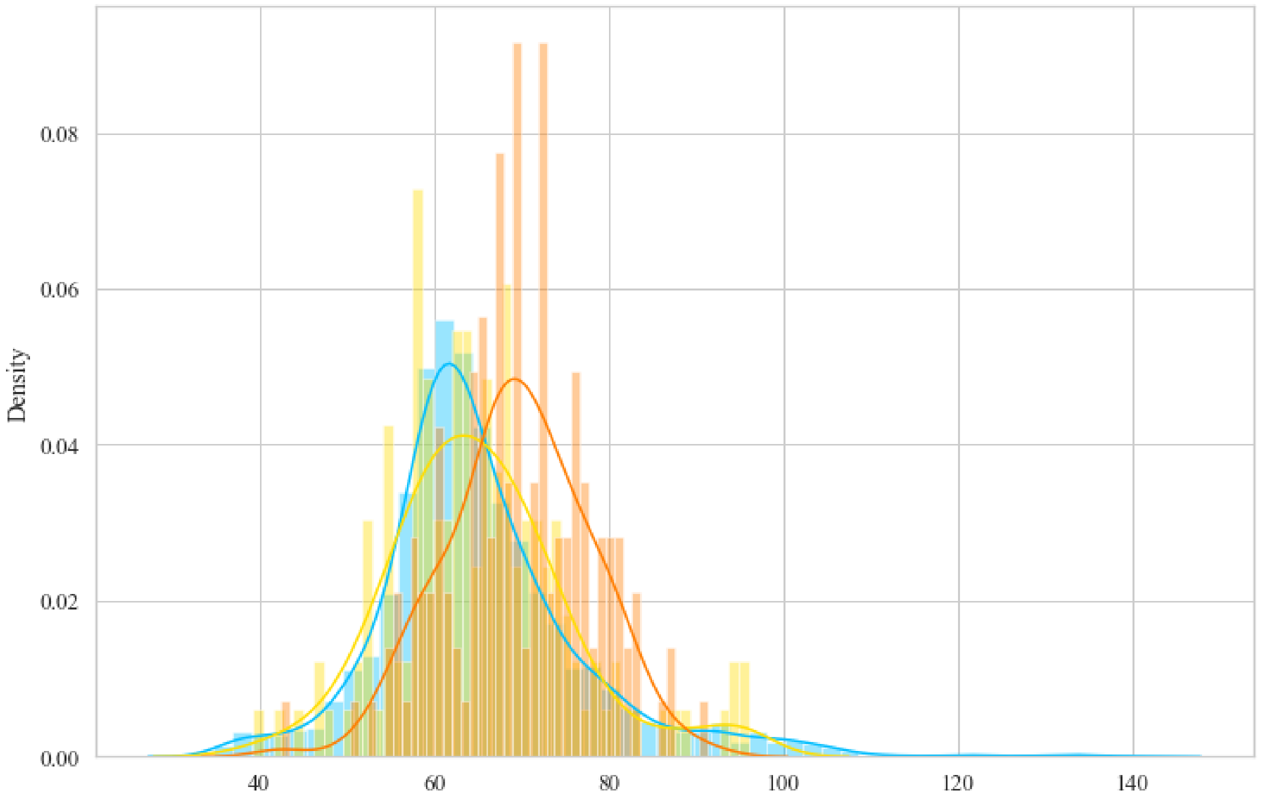
Using the threshold, we can turn the problem into a simple binary classification task:

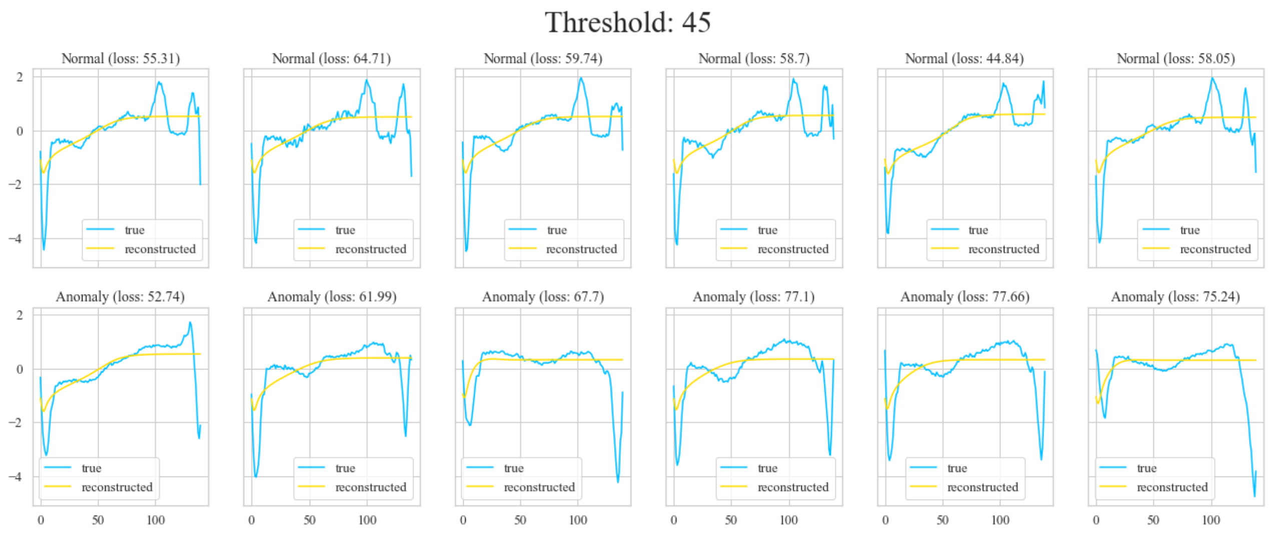
- If the reconstruction loss for an example is below the threshold, we'll classify it as a *normal* heartbeat
- Alternatively, if the loss is higher than the threshold, we'll classify it as an anomaly

In [25]:
```
_, losses = predict(model, train_dataset)
sns.distplot(losses, bins=50, kde=True);
THRESHOLD = 45

# Evaluation
# Using the threshold, we can turn the problem into a simple binary classification task
# - If the reconstruction loss for an example is below the threshold, we'll classify it
# - Alternatively, if the loss is higher than the threshold, we'll classify it as an


# Normal hearbeats
# Let's check how well our model does on normal heartbeats. We'll use the normal heartb
predictions, pred_losses = predict(model, test_normal_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

# We'll count the correct predictions:
correct = sum(l <= THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(test_normal_dataset))*100, 3)
embedding_dimension_2_normal_recall = recall_percentage
print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)} ({round(recall


# Anomalies
#We'll do the same with the anomaly examples, but their number is much higher. We'll ge
anomaly_dataset = test_anomaly_dataset[:len(test_normal_dataset)]
# anomaly_dataset = test_anomaly_dataset
"""Now we can take the predictions of our model for the subset of anomalies:"""

predictions, pred_losses = predict(model, anomaly_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

"""Finally, we can count the number of examples above the threshold (considered as anom

correct = sum(l > THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(anomaly_dataset))*100, 3)
embedding_dimension_2_anomaly_recall = recall_percentage
print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)} ({round(recall_pe


# Look at Examples

# We can overlay the real and reconstructed Time Series values to see how close they ar
# We'll do it for some normal and anomaly cases:
def plot_prediction(data, model, title, ax):
    predictions, pred_losses = predict(model, [data])

    ax.plot(data, label='true')
    ax.plot(predictions[0], label='reconstructed')
    ax.set_title(f'{title} (loss: {np.around(pred_losses[0], 2)})')
    ax.legend()


fig, axs = plt.subplots(
  nrows=2,
  ncols=6,
  sharey=True,
  sharex=True,
  figsize=(22, 8))

for i, data in enumerate(test_normal_dataset[:6]):
    plot_prediction(data, model, title='Normal', ax=axs[0, i])

for i, data in enumerate(test_anomaly_dataset[:6]):
    plot_prediction(data, model, title='Anomaly', ax=axs[1, i])
```

```
fig.suptitle('Threshold: ' + str(THRESHOLD), fontsize=30)
plt.show()
```

```
Correct normal predictions: 3/145 (2.07%)
Correct anomaly predictions: 144/145 (99.31%)
```





Embedding Dimension 4

In [53]:
```
# Our Autoencoder passes the input through the Encoder and Decoder. Let's create an ins

model = RecurrentAutoencoder(seq_len, n_features, 4)
model = model.to(device)
```

At each epoch, the training process feeds our model with all training examples and evaluates the performance on the validation set. Note that we're using a batch size of 1 (our model sees only 1 sequence at a time). We also record the training and validation set losses during the process.

Note that we're minimizing the L1Loss, which measures the MAE (mean absolute error). Why? The reconstructions seem to be better than with MSE (mean squared error).

We'll get the version of the model with the smallest validation error. Let's do some training:

In [54]:
```
model, history = train_model(
    model,
    train_dataset,
    val_dataset,
    n_epochs=25)
embedding_dimension_4_train_loss = history['train'].copy()
embedding_dimension_4_val_loss = history['val'].copy()
```

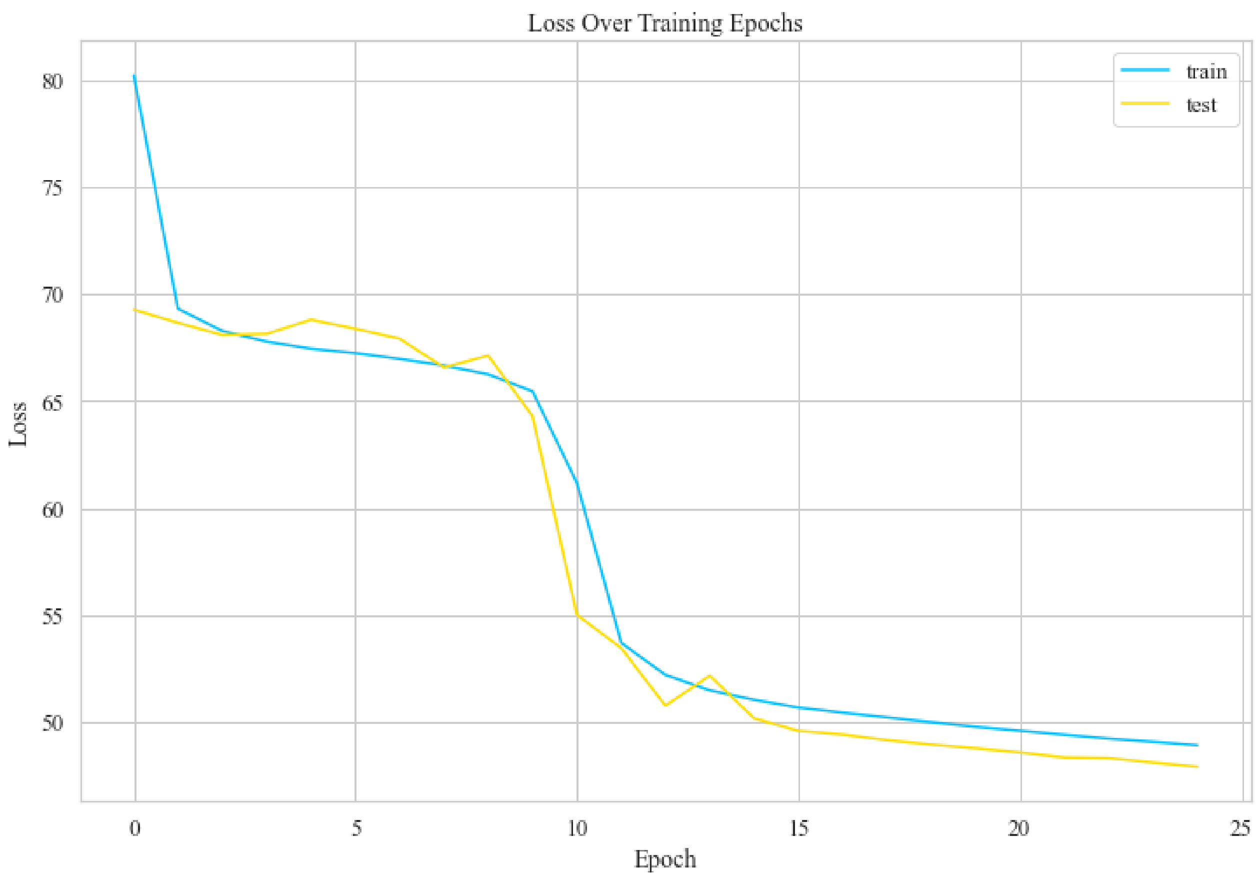
```
Epoch 1: train loss 80.19878390680057 val loss 69.26908168857821
Epoch 2: train loss 69.31571875421908 val loss 68.65745803520541
Epoch 3: train loss 68.27495130133792 val loss 68.08729091032373
Epoch 4: train loss 67.7798405408571 val loss 68.14061145977762
Epoch 5: train loss 67.44669699524738 val loss 68.80219272704662
Epoch 6: train loss 67.2384648198132 val loss 68.37944974671451
Epoch 7: train loss 66.96409542817159 val loss 67.91853432769254
Epoch 8: train loss 66.65306984787264 val loss 66.56333002865111
Epoch 9: train loss 66.24268106893396 val loss 67.12109032588607
Epoch 10: train loss 65.45253648338948 val loss 64.28960840531177
Epoch 11: train loss 61.17639161386302 val loss 55.01288908089387
Epoch 12: train loss 53.70907468538234 val loss 53.46417814391465
Epoch 13: train loss 52.216819098955774 val loss 50.77934258626996
Epoch 14: train loss 51.49541086844983 val loss 52.18187155088874
Epoch 15: train loss 51.05408894924424 val loss 50.176387344204116
Epoch 16: train loss 50.6866795633263 val loss 49.5953996287271
Epoch 17: train loss 50.44328287963375 val loss 49.422665631811775
Epoch 18: train loss 50.22289363302948 val loss 49.162473008087474
Epoch 19: train loss 49.998825981942744 val loss 48.95537481210338
Epoch 20: train loss 49.78346392821035 val loss 48.78194568100236
Epoch 21: train loss 49.59675631355922 val loss 48.586259002164766
Epoch 22: train loss 49.4135122141594 val loss 48.35374547027484
Epoch 23: train loss 49.23241297096458 val loss 48.322291403500294
Epoch 24: train loss 49.0851545391521 val loss 48.1181082090827
Epoch 25: train loss 48.92101011618353 val loss 47.91901927596473
```

Choosing a Threshold

In [55]:
```
def predict(model, dataset):
    """The reconstruction error on the training set. Let's start by writing a helper fu
    predictions from our model"""
    predictions, losses = [], []
    criterion = nn.L1Loss(reduction='sum').to(device)
    with torch.no_grad():
        model = model.eval()
        for seq_true in dataset:
            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            predictions.append(seq_pred.cpu().numpy().flatten())
            losses.append(loss.item())
    return predictions, losses
```

In [56]:

```
ax = plt.figure().gca()

ax.plot(history['train'])
ax.plot(history['val'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'])
plt.title('Loss Over Training Epochs')

plt.show()
```



Save the Model

In [57]:

```
MODEL_PATH = 'model25_4.pth'
torch.save(model, MODEL_PATH)
```

Load the Model

In [58]:

```
MODEL_PATH = 'model25_4.pth'
model = torch.load(MODEL_PATH)
```

Evaluation

Using the threshold, we can turn the problem into a simple binary classification task:

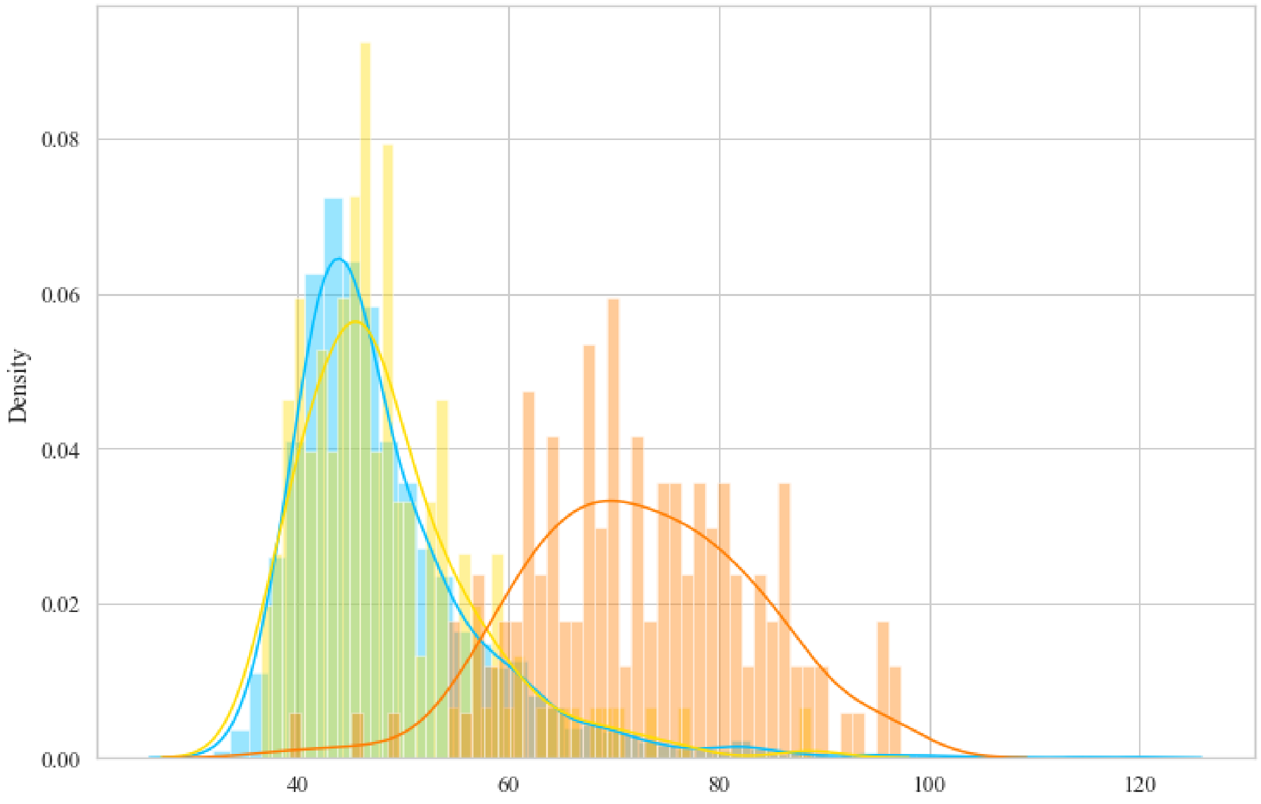
- If the reconstruction loss for an example is below the threshold, we'll classify it as a *normal* heartbeat
- Alternatively, if the loss is higher than the threshold, we'll classify it as an anomaly

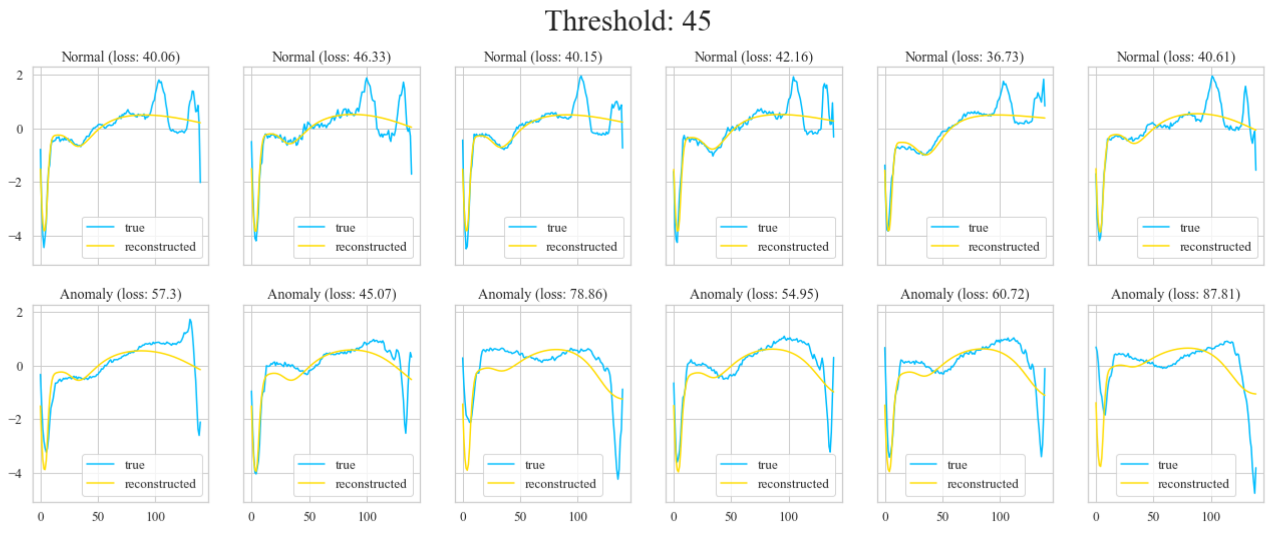
In [59]:

```
_, losses = predict(model, train_dataset)
sns.distplot(losses, bins=50, kde=True);
THRESHOLD = 45

# Evaluation
# Using the threshold, we can turn the problem into a simple binary classification task
# - If the reconstruction loss for an example is below the threshold, we'll classify it
# - Alternatively, if the loss is higher than the threshold, we'll classify it as an an


# Normal hearbeats
# Let's check how well our model does on normal heartbeats. We'll use the normal heartb
predictions, pred_losses = predict(model, test_normal_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

# We'll count the correct predictions:
correct = sum(l <= THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(test_normal_dataset))*100, 3)
embedding_dimension_4_normal_recall = recall_percentage
print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)} ({round(recall


# Anomalies
#We'll do the same with the anomaly examples, but their number is much higher. We'll ge
anomaly_dataset = test_anomaly_dataset[:len(test_normal_dataset)]
# anomaly_dataset = test_anomaly_dataset
"""Now we can take the predictions of our model for the subset of anomalies:"""

predictions, pred_losses = predict(model, anomaly_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

"""Finally, we can count the number of examples above the threshold (considered as anom

correct = sum(l > THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(anomaly_dataset))*100, 3)
embedding_dimension_4_anomaly_recall = recall_percentage
print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)} ({round(recall_pe


# Look at Examples

# We can overlay the real and reconstructed Time Series values to see how close they ar
# We'll do it for some normal and anomaly cases:
def plot_prediction(data, model, title, ax):
    predictions, pred_losses = predict(model, [data])

    ax.plot(data, label='true')
    ax.plot(predictions[0], label='reconstructed')
    ax.set_title(f'{title} (loss: {np.around(pred_losses[0], 2)})')
    ax.legend()
```

```
fig, axs = plt.subplots(
  nrows=2,
  ncols=6,
  sharey=True,
  sharex=True,
  figsize=(22, 8))

for i, data in enumerate(test_normal_dataset[:6]):
    plot_prediction(data, model, title='Normal', ax=axs[0, i])

for i, data in enumerate(test_anomaly_dataset[:6]):
    plot_prediction(data, model, title='Anomaly', ax=axs[1, i])

fig.suptitle('Threshold: ' + str(THRESHOLD), fontsize=30)
plt.show()
```

```
Correct normal predictions: 52/145 (35.86%)
Correct anomaly predictions: 144/145 (99.31%)
```





Embedding Dimension 6

In [33]:
```
# Our Autoencoder passes the input through the Encoder and Decoder. Let's create an ins

model = RecurrentAutoencoder(seq_len, n_features, 6)
model = model.to(device)
```

At each epoch, the training process feeds our model with all training examples and evaluates the performance on the validation set. Note that we're using a batch size of 1 (our model sees only 1 sequence at a time). We also record the training and validation set losses during the process.

Note that we're minimizing the L1Loss, which measures the MAE (mean absolute error). Why? The reconstructions seem to be better than with MSE (mean squared error).

We'll get the version of the model with the smallest validation error. Let's do some training:

In [34]:
```
model, history = train_model(
    model,
    train_dataset,
    val_dataset,
    n_epochs=25)
embedding_dimension_6_train_loss = history['train'].copy()
embedding_dimension_6_val_loss = history['val'].copy()
```

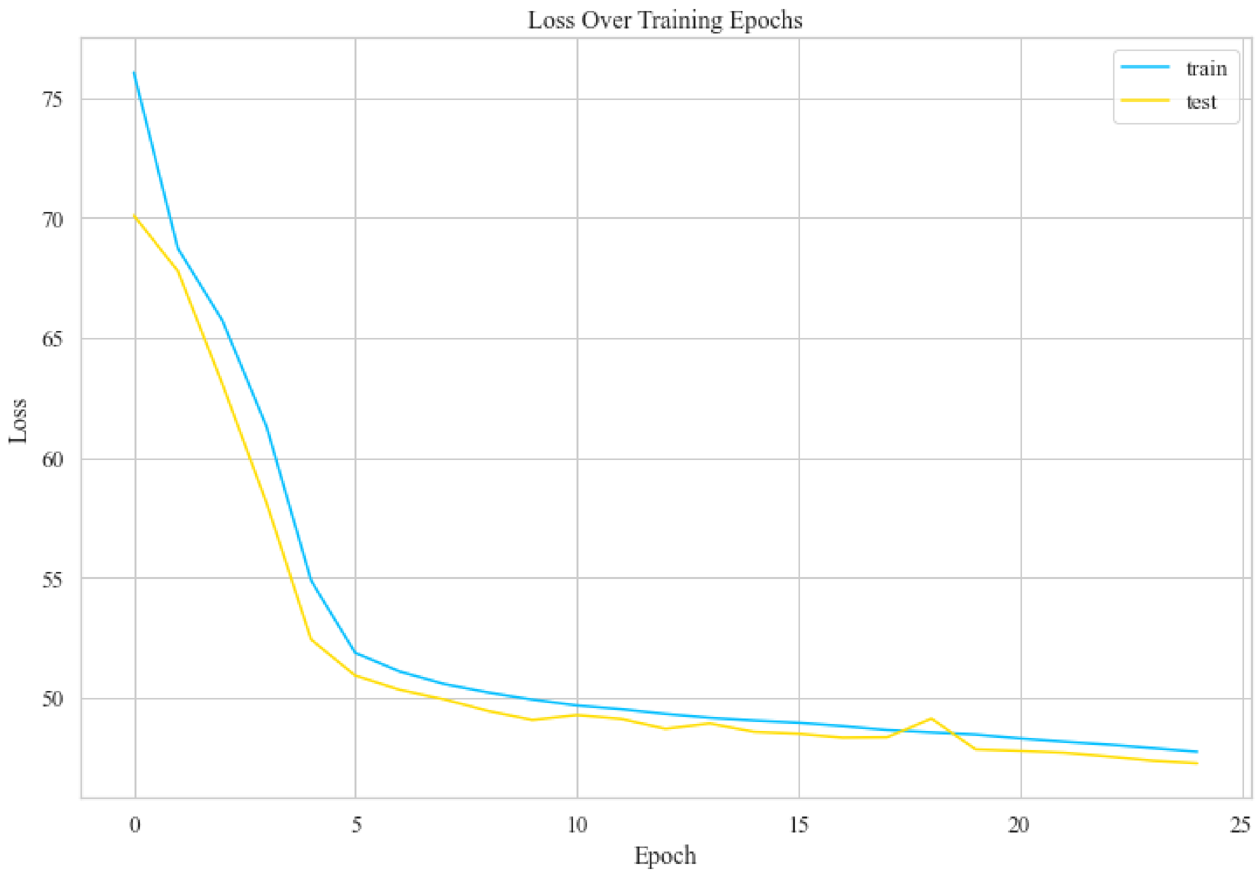
```
Epoch 1: train loss 76.04306476628382 val loss 70.08952845811031
Epoch 2: train loss 68.71593424909685 val loss 67.77663767948086
Epoch 3: train loss 65.72611671549234 val loss 63.080240360299065
Epoch 4: train loss 61.281167272885256 val loss 58.107157215730325
Epoch 5: train loss 54.89719512490484 val loss 52.44121209102279
Epoch 6: train loss 51.88142259944499 val loss 50.936188655501745
Epoch 7: train loss 51.10630791036413 val loss 50.3445593537731
Epoch 8: train loss 50.58551119412688 val loss 49.94222618858155
Epoch 9: train loss 50.23087372293591 val loss 49.45390071152827
Epoch 10: train loss 49.93069092067486 val loss 49.07733432912989
Epoch 11: train loss 49.685586196671856 val loss 49.28430963457648
Epoch 12: train loss 49.526938550273336 val loss 49.12212352297005
Epoch 13: train loss 49.33122578766595 val loss 48.7139276237618
Epoch 14: train loss 49.168217907697624 val loss 48.932773974161506
Epoch 15: train loss 49.05497338165443 val loss 48.58398692030142
Epoch 16: train loss 48.96272835583516 val loss 48.5050441234185
Epoch 17: train loss 48.82447220519011 val loss 48.341784526056806
Epoch 18: train loss 48.66151985435993 val loss 48.35958384491071
Epoch 19: train loss 48.56184193397039 val loss 49.1341879717726
Epoch 20: train loss 48.47064090899429 val loss 47.85862068592892
Epoch 21: train loss 48.31016150601013 val loss 47.79081496931994
Epoch 22: train loss 48.17505987398762 val loss 47.712234379895314
Epoch 23: train loss 48.052926806565594 val loss 47.55747889495954
Epoch 24: train loss 47.90886135939291 val loss 47.38433082762839
Epoch 25: train loss 47.755415047334594 val loss 47.27618195986178
```

Choosing a Threshold

In [35]:
```
def predict(model, dataset):
    """The reconstruction error on the training set. Let's start by writing a helper fu
    predictions from our model"""
    predictions, losses = [], []
    criterion = nn.L1Loss(reduction='sum').to(device)
    with torch.no_grad():
        model = model.eval()
        for seq_true in dataset:
            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            predictions.append(seq_pred.cpu().numpy().flatten())
            losses.append(loss.item())
    return predictions, losses
```

In [36]:

```
ax = plt.figure().gca()

ax.plot(history['train'])
ax.plot(history['val'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'])
plt.title('Loss Over Training Epochs')

plt.show()
```



Save the Model

In [37]:

```
MODEL_PATH = 'model25_6.pth'
torch.save(model, MODEL_PATH)
```

Load the Model

In [38]:
```
MODEL_PATH = 'model25_6.pth'
model = torch.load(MODEL_PATH)
```

Evaluation

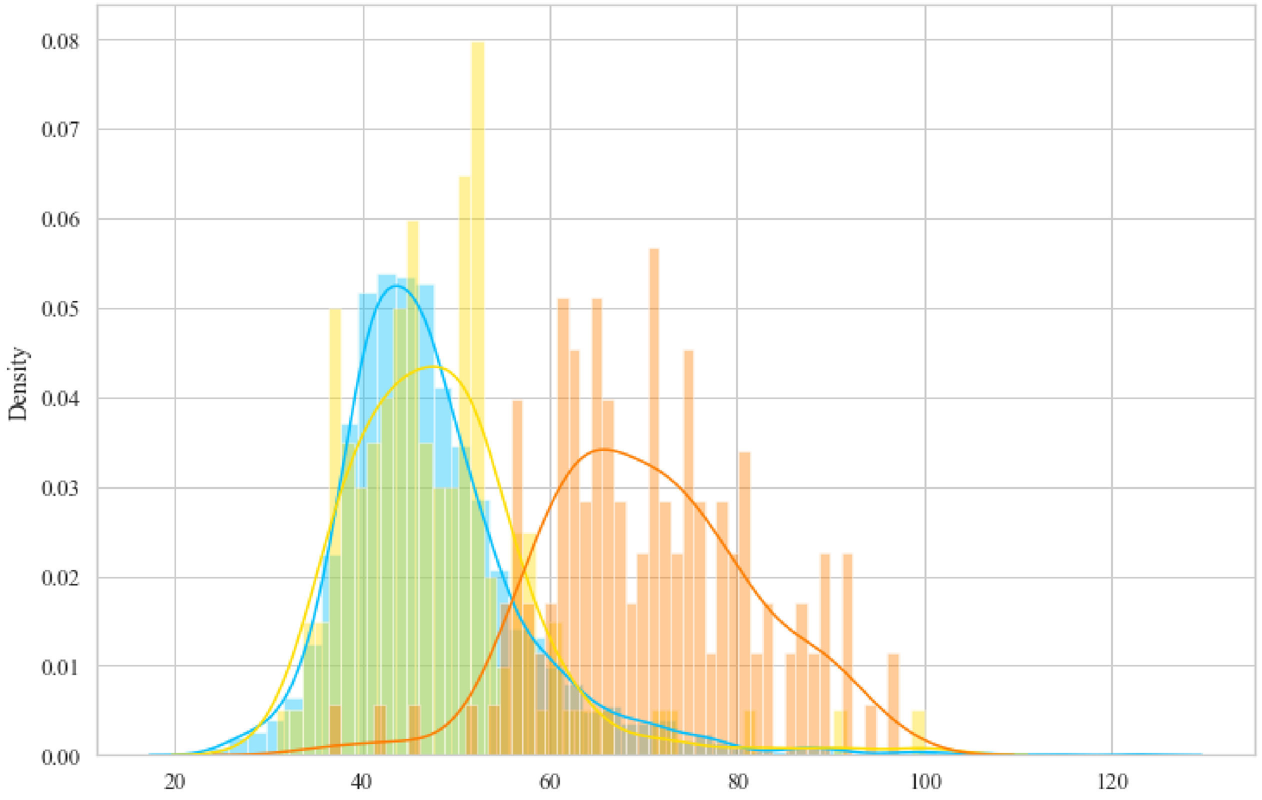
Using the threshold, we can turn the problem into a simple binary classification task:

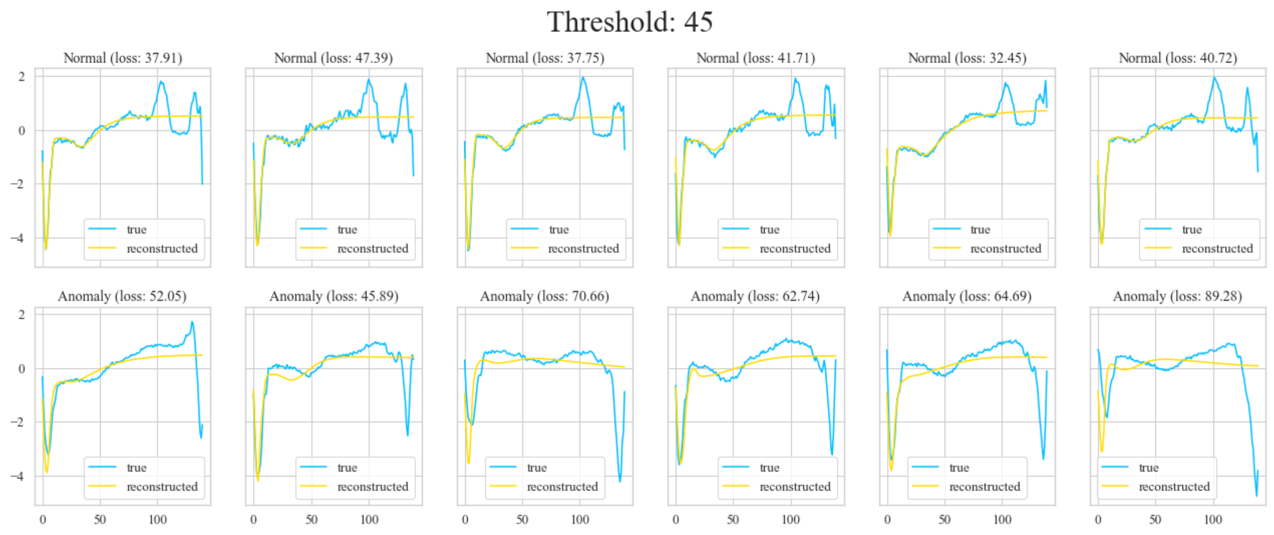
- If the reconstruction loss for an example is below the threshold, we'll classify it as a *normal* heartbeat
- Alternatively, if the loss is higher than the threshold, we'll classify it as an anomaly

In [39]:
```
_, losses = predict(model, train_dataset)
sns.distplot(losses, bins=50, kde=True);
THRESHOLD = 45

# Evaluation
# Using the threshold, we can turn the problem into a simple binary classification task
# - If the reconstruction loss for an example is below the threshold, we'll classify it
# - Alternatively, if the loss is higher than the threshold, we'll classify it as an an


# Normal hearbeats
# Let's check how well our model does on normal heartbeats. We'll use the normal heartb
predictions, pred_losses = predict(model, test_normal_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

# We'll count the correct predictions:
correct = sum(l <= THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(test_normal_dataset))*100, 3)
embedding_dimension_6_normal_recall = recall_percentage
print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)} ({round(recall


# Anomalies
#We'll do the same with the anomaly examples, but their number is much higher. We'll ge
anomaly_dataset = test_anomaly_dataset[:len(test_normal_dataset)]
# anomaly_dataset = test_anomaly_dataset
"""Now we can take the predictions of our model for the subset of anomalies:"""

predictions, pred_losses = predict(model, anomaly_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

"""Finally, we can count the number of examples above the threshold (considered as anom

correct = sum(l > THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(anomaly_dataset))*100, 3)
embedding_dimension_6_anomaly_recall = recall_percentage
print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)} ({round(recall_pe


# Look at Examples
```

```python
# We can overlay the real and reconstructed Time Series values to see how close they ar
# We'll do it for some normal and anomaly cases:
def plot_prediction(data, model, title, ax):
    predictions, pred_losses = predict(model, [data])

    ax.plot(data, label='true')
    ax.plot(predictions[0], label='reconstructed')
    ax.set_title(f'{title} (loss: {np.around(pred_losses[0], 2)})')
    ax.legend()


fig, axs = plt.subplots(
  nrows=2,
  ncols=6,
  sharey=True,
  sharex=True,
  figsize=(22, 8))

for i, data in enumerate(test_normal_dataset[:6]):
    plot_prediction(data, model, title='Normal', ax=axs[0, i])

for i, data in enumerate(test_anomaly_dataset[:6]):
    plot_prediction(data, model, title='Anomaly', ax=axs[1, i])

fig.suptitle('Threshold: ' + str(THRESHOLD), fontsize=30)
plt.show()
```

```
Correct normal predictions: 59/145 (40.69%)
Correct anomaly predictions: 143/145 (98.62%)
```





Embedding Dimension 8

In [40]:

```
# Our Autoencoder passes the input through the Encoder and Decoder. Let's create an ins

model = RecurrentAutoencoder(seq_len, n_features, 8)
model = model.to(device)
```

At each epoch, the training process feeds our model with all training examples and evaluates the performance on the validation set. Note that we're using a batch size of 1 (our model sees only 1 sequence at a time). We also record the training and validation set losses during the process.

Note that we're minimizing the L1Loss, which measures the MAE (mean absolute error). Why? The reconstructions seem to be better than with MSE (mean squared error).

We'll get the version of the model with the smallest validation error. Let's do some training:

In [41]:

```
model, history = train_model(
    model,
    train_dataset,
    val_dataset,
    n_epochs=25)
embedding_dimension_8_train_loss = history['train'].copy()
embedding_dimension_8_val_loss = history['val'].copy()
```

```
Epoch 1: train loss 73.37781162715544 val loss 64.7008149469265
Epoch 2: train loss 57.565313845860864 val loss 53.620411270714456
Epoch 3: train loss 53.11115357915224 val loss 51.77878189086914
Epoch 4: train loss 51.68009333243441 val loss 50.16512059026204
Epoch 5: train loss 46.526802969382494 val loss 47.335399256631376
Epoch 6: train loss 39.25469285180039 val loss 39.442831000370376
Epoch 7: train loss 36.64589412478563 val loss 35.05569817669969
Epoch 8: train loss 35.30766740819708 val loss 37.39315108471763
Epoch 9: train loss 34.96584619355269 val loss 37.96691188877353
Epoch 10: train loss 34.10409068254058 val loss 32.58600108045767
Epoch 11: train loss 33.81211978338842 val loss 36.61893816612686
Epoch 12: train loss 33.40794637211865 val loss 33.72315236244592
Epoch 13: train loss 32.92189261472211 val loss 33.97583022133889
Epoch 14: train loss 32.42911179591359 val loss 33.73300271148161
Epoch 15: train loss 32.09755123588934 val loss 33.1068570556901
Epoch 16: train loss 31.871542454726846 val loss 32.716613411496525
Epoch 17: train loss 31.744264863278296 val loss 33.068462723351175
Epoch 18: train loss 31.432315177948233 val loss 33.347710951196454
Epoch 19: train loss 31.40974400323132 val loss 30.977443701578082
Epoch 20: train loss 30.99963922089312 val loss 31.490810439855167
Epoch 21: train loss 30.647773671947817 val loss 31.199090153690896
Epoch 22: train loss 30.565437132386805 val loss 30.84645486122105
Epoch 23: train loss 30.264405988195836 val loss 30.999527049146007
Epoch 24: train loss 30.042432734294554 val loss 30.966294828942203
Epoch 25: train loss 29.912456299882894 val loss 30.614160534464865
```

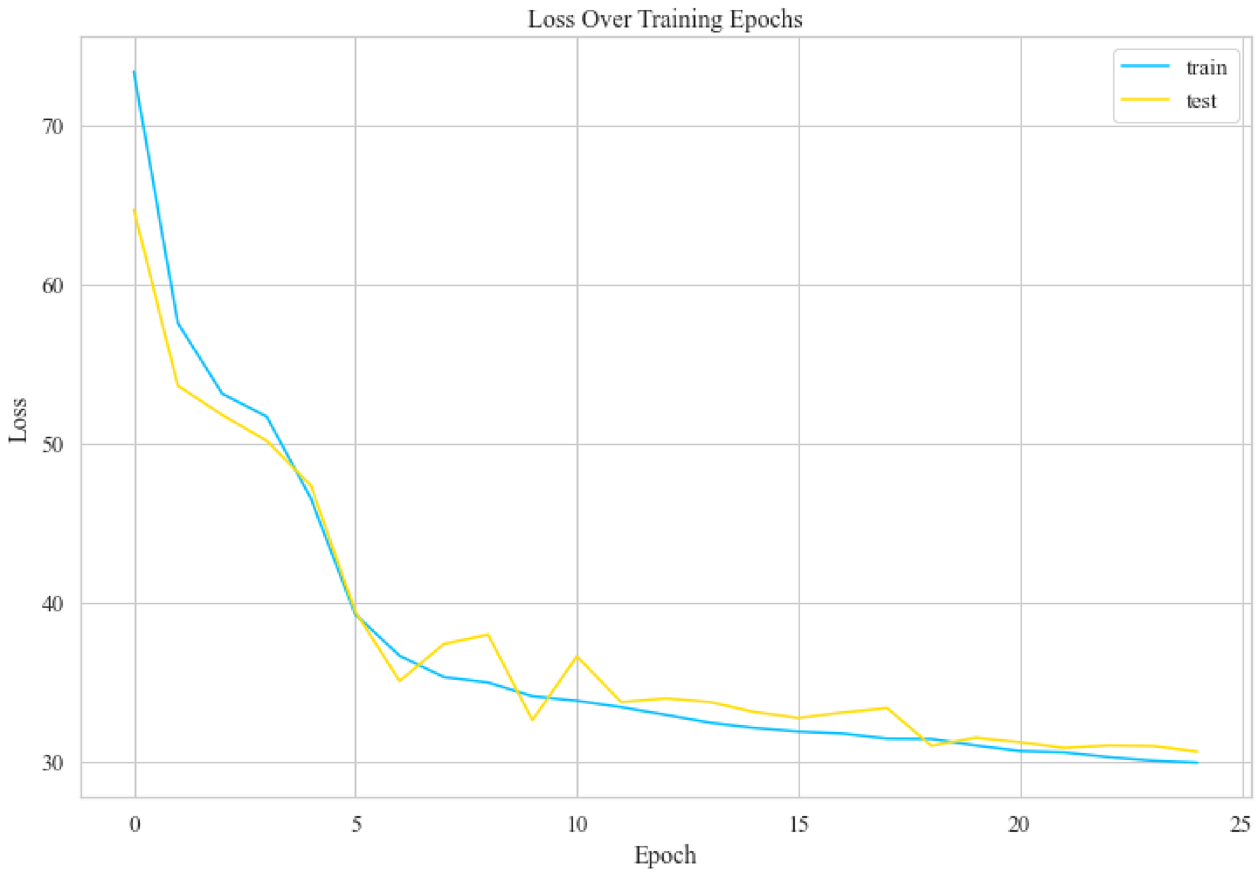
Choosing a Threshold

In [42]:

```python
def predict(model, dataset):
    """The reconstruction error on the training set. Let's start by writing a helper fu
    predictions from our model"""
    predictions, losses = [], []
    criterion = nn.L1Loss(reduction='sum').to(device)
    with torch.no_grad():
        model = model.eval()
        for seq_true in dataset:
            seq_true = seq_true.to(device)
            seq_pred = model(seq_true)

            loss = criterion(seq_pred, seq_true)

            predictions.append(seq_pred.cpu().numpy().flatten())
            losses.append(loss.item())
    return predictions, losses
```

In [43]:

```python
ax = plt.figure().gca()

ax.plot(history['train'])
ax.plot(history['val'])
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'])
plt.title('Loss Over Training Epochs')

plt.show()
```



Save the Model

In [44]:

```
MODEL_PATH = 'model25_8.pth'
torch.save(model, MODEL_PATH)
```

Load the Model

In [45]:

```
MODEL_PATH = 'model25_8.pth'
model = torch.load(MODEL_PATH)
```

Evaluation

Using the threshold, we can turn the problem into a simple binary classification task:

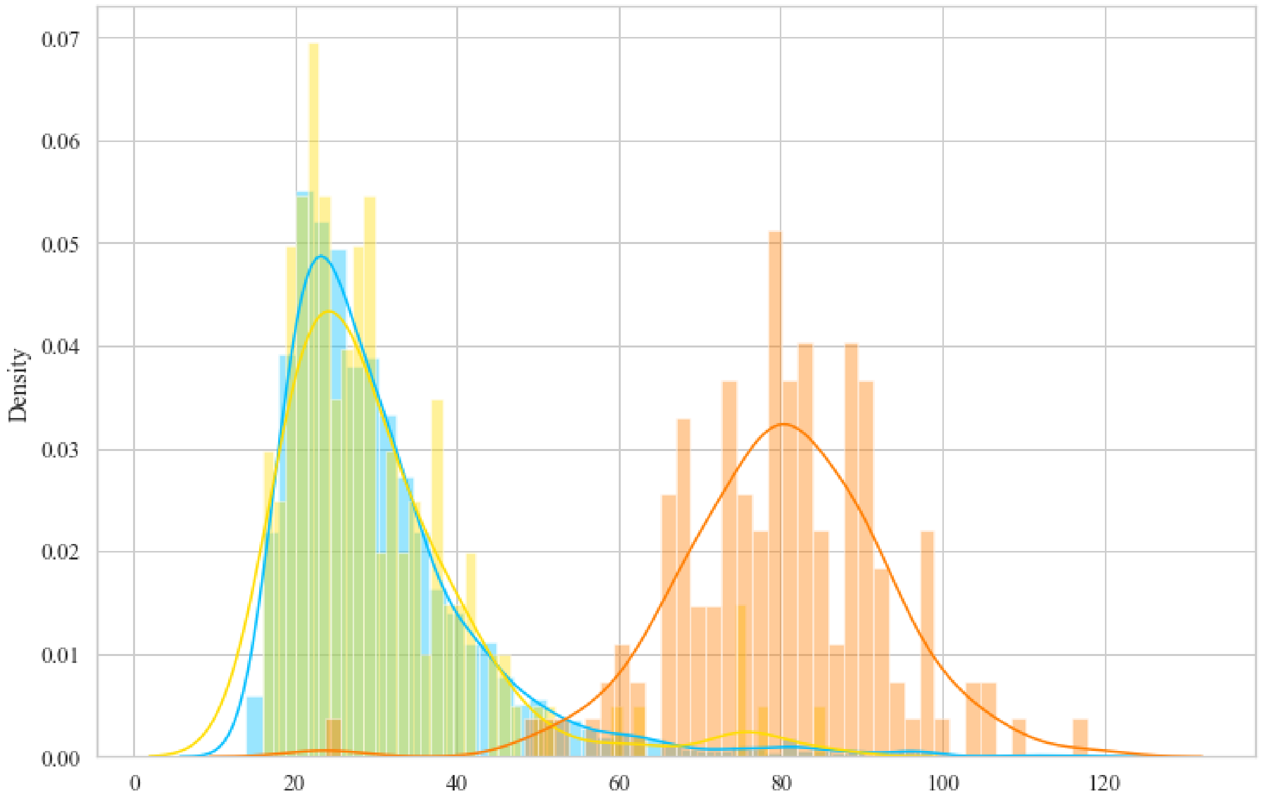
- If the reconstruction loss for an example is below the threshold, we'll classify it as a *normal* heartbeat
- Alternatively, if the loss is higher than the threshold, we'll classify it as an anomaly

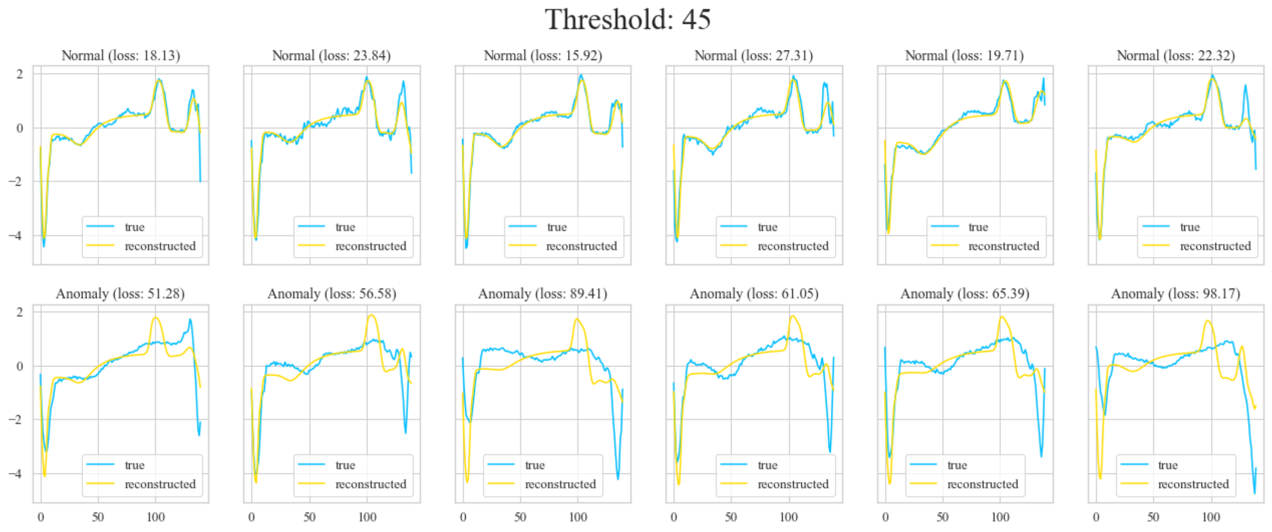
In [46]:

```
_, losses = predict(model, train_dataset)
sns.distplot(losses, bins=50, kde=True);
THRESHOLD = 45

# Evaluation
# Using the threshold, we can turn the problem into a simple binary classification task
# - If the reconstruction loss for an example is below the threshold, we'll classify it
# - Alternatively, if the loss is higher than the threshold, we'll classify it as an an


# Normal hearbeats
# Let's check how well our model does on normal heartbeats. We'll use the normal heartb
predictions, pred_losses = predict(model, test_normal_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

# We'll count the correct predictions:
correct = sum(l <= THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(test_normal_dataset))*100, 3)
embedding_dimension_8_normal_recall = recall_percentage
print(f'Correct normal predictions: {correct}/{len(test_normal_dataset)} ({round(recall


# Anomalies
#We'll do the same with the anomaly examples, but their number is much higher. We'll ge
anomaly_dataset = test_anomaly_dataset[:len(test_normal_dataset)]
# anomaly_dataset = test_anomaly_dataset
"""Now we can take the predictions of our model for the subset of anomalies:"""

predictions, pred_losses = predict(model, anomaly_dataset)
sns.distplot(pred_losses, bins=50, kde=True);

"""Finally, we can count the number of examples above the threshold (considered as anom

correct = sum(l > THRESHOLD for l in pred_losses)
recall_percentage = round((correct/len(anomaly_dataset))*100, 3)
embedding_dimension_8_anomaly_recall = recall_percentage
print(f'Correct anomaly predictions: {correct}/{len(anomaly_dataset)} ({round(recall_pe


# Look at Examples

# We can overlay the real and reconstructed Time Series values to see how close they ar
# We'll do it for some normal and anomaly cases:
def plot_prediction(data, model, title, ax):
    predictions, pred_losses = predict(model, [data])

    ax.plot(data, label='true')
    ax.plot(predictions[0], label='reconstructed')
    ax.set_title(f'{title} (loss: {np.around(pred_losses[0], 2)})')
    ax.legend()


fig, axs = plt.subplots(
  nrows=2,
  ncols=6,
  sharey=True,
  sharex=True,
  figsize=(22, 8))

for i, data in enumerate(test_normal_dataset[:6]):
    plot_prediction(data, model, title='Normal', ax=axs[0, i])

for i, data in enumerate(test_anomaly_dataset[:6]):
    plot_prediction(data, model, title='Anomaly', ax=axs[1, i])
```

```
fig.suptitle('Threshold: ' + str(THRESHOLD), fontsize=30)
plt.show()
```

```
Correct normal predictions: 133/145 (91.72%)
Correct anomaly predictions: 144/145 (99.31%)
```





Pickle Dump

In [47]:
```
pickle.dump(embedding_dimension_2_train_loss, open('embedding_dimension_2_train_loss.p'
pickle.dump(embedding_dimension_2_val_loss, open('embedding_dimension_2_val_loss.p', 'w
pickle.dump(embedding_dimension_2_normal_recall, open('embedding_dimension_2_normal_rec
pickle.dump(embedding_dimension_2_anomaly_recall, open('embedding_dimension_2_anomaly_r

pickle.dump(embedding_dimension_4_train_loss, open('embedding_dimension_4_train_loss.p'
pickle.dump(embedding_dimension_4_val_loss, open('embedding_dimension_4_val_loss.p', 'w
pickle.dump(embedding_dimension_4_normal_recall, open('embedding_dimension_4_normal_rec
pickle.dump(embedding_dimension_4_anomaly_recall, open('embedding_dimension_4_anomaly_r
pickle.dump(embedding_dimension_6_train_loss, open('embedding_dimension_6_train_loss.p'
pickle.dump(embedding_dimension_6_val_loss, open('embedding_dimension_6_val_loss.p', 'w
pickle.dump(embedding_dimension_6_normal_recall, open('embedding_dimension_6_normal_rec
pickle.dump(embedding_dimension_6_anomaly_recall, open('embedding_dimension_6_anomaly_r

pickle.dump(embedding_dimension_8_train_loss, open('embedding_dimension_8_train_loss.p'
pickle.dump(embedding_dimension_8_val_loss, open('embedding_dimension_8_val_loss.p', 'w
pickle.dump(embedding_dimension_8_normal_recall, open('embedding_dimension_8_normal_rec
pickle.dump(embedding_dimension_8_anomaly_recall, open('embedding_dimension_8_anomaly_r
```

In [3]:

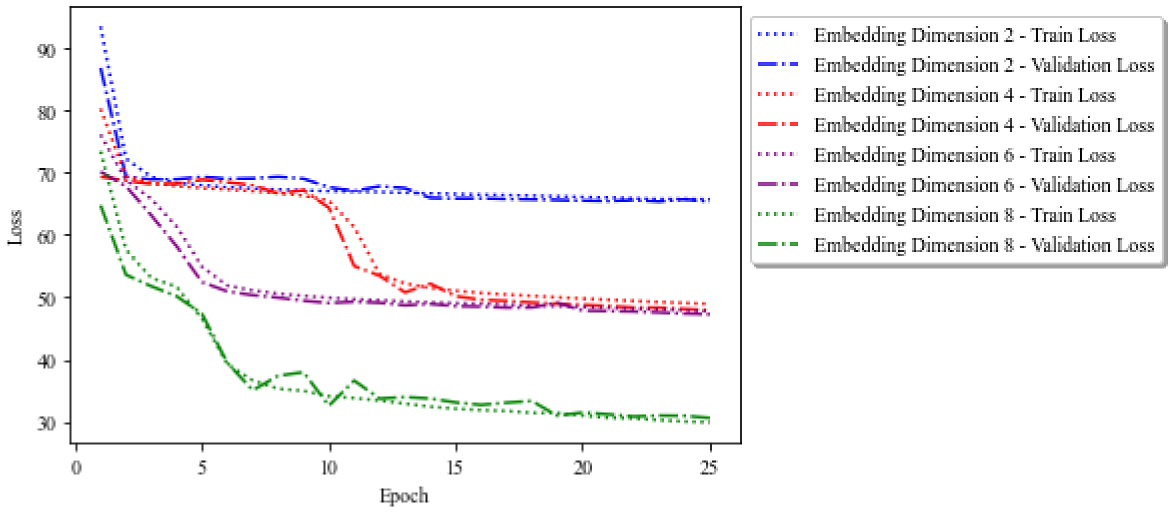
```
embedding_dimension_2_train_loss = pickle.load(open('embedding_dimension_2_train_loss.p
embedding_dimension_2_val_loss = pickle.load(open('embedding_dimension_2_val_loss.p', '
embedding_dimension_2_normal_recall = pickle.load(open('embedding_dimension_2_normal_re
embedding_dimension_2_anomaly_recall = pickle.load(open('embedding_dimension_2_anomaly_

embedding_dimension_4_train_loss = pickle.load(open('embedding_dimension_4_train_loss.p
embedding_dimension_4_val_loss = pickle.load(open('embedding_dimension_4_val_loss.p', '
embedding_dimension_4_normal_recall = pickle.load(open('embedding_dimension_4_normal_re
embedding_dimension_4_anomaly_recall = pickle.load(open('embedding_dimension_4_anomaly_

embedding_dimension_6_train_loss = pickle.load(open('embedding_dimension_6_train_loss.p
embedding_dimension_6_val_loss = pickle.load(open('embedding_dimension_6_val_loss.p', '
embedding_dimension_6_normal_recall = pickle.load(open('embedding_dimension_6_normal_re
embedding_dimension_6_anomaly_recall = pickle.load(open('embedding_dimension_6_anomaly_

embedding_dimension_8_train_loss = pickle.load(open('embedding_dimension_8_train_loss.p
embedding_dimension_8_val_loss = pickle.load(open('embedding_dimension_8_val_loss.p', '
embedding_dimension_8_normal_recall = pickle.load(open('embedding_dimension_8_normal_re
embedding_dimension_8_anomaly_recall = pickle.load(open('embedding_dimension_8_anomaly_
```

b) Briefly explain the trend you see in the training and validation loss.

In [24]:

```
fig, ax = plt.subplots()

epoch_label = np.arange(1, 26, 1)
ax.plot(epoch_label, embedding_dimension_2_train_loss, label='Embedding Dimension 2 - T
        linestyle='dotted', color='blue')
ax.plot(epoch_label, embedding_dimension_2_val_loss, label='Embedding Dimension 2 - Val
        linestyle='dashdot', color='blue')

ax.plot(epoch_label, embedding_dimension_4_train_loss, label='Embedding Dimension 4 - T
        linestyle='dotted', color='red')
ax.plot(epoch_label, embedding_dimension_4_val_loss, label='Embedding Dimension 4 - Val
        linestyle='dashdot', color='red')

ax.plot(epoch_label, embedding_dimension_6_train_loss, label='Embedding Dimension 6 - T
        linestyle='dotted', color='purple')
ax.plot(epoch_label, embedding_dimension_6_val_loss, label='Embedding Dimension 6 - Val
        linestyle='dashdot', color='purple')

ax.plot(epoch_label, embedding_dimension_8_train_loss, label='Embedding Dimension 8 - T
        linestyle='dotted', color='green')
ax.plot(epoch_label, embedding_dimension_8_val_loss, label='Embedding Dimension 8 - Val
        linestyle='dashdot', color='green')
```

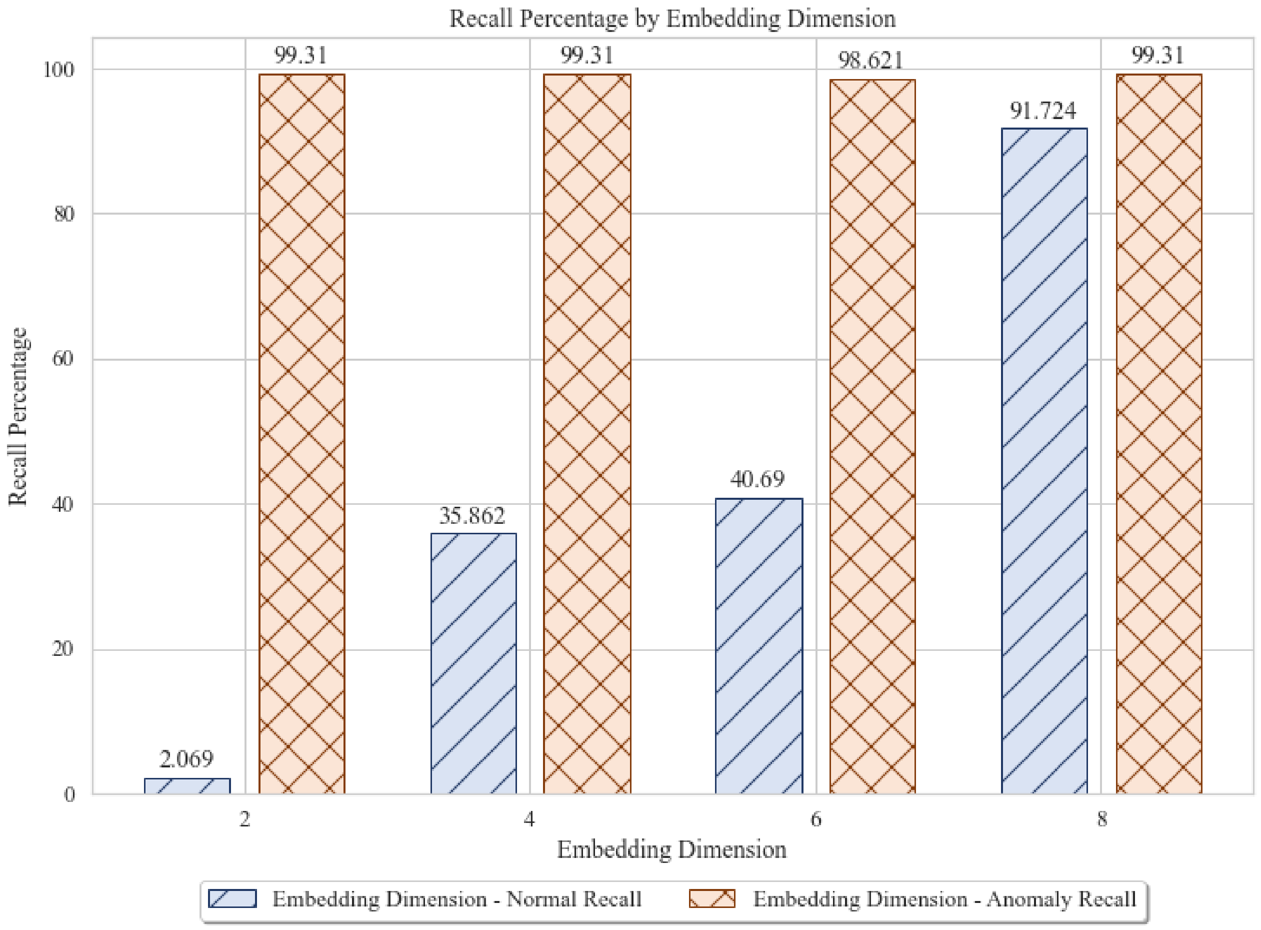
```
ax.legend(loc='upper left', bbox_to_anchor=(1, 1), fancybox=True, shadow=True, ncol=1)
ax.set_ylabel('Loss')
ax.set_xlabel('Epoch')

plt.show()
```



When the epoch level is fixed, as the embedding dimension is increased, the loss is significantly decreased. As the embedding dimension is increased, in the bottle neck structure, since the encoder does not reduce the array to a small dimension, the decoder does not produce significant loss.

c) Compute the proportion of normal and abnormal time-series correctly classified (i.e. *Recall*) for the same test set in Q.1 above for each of the embedding dimension values from (a). You can set the threshold to 45.

In [63]:
```
fig, ax = plt.subplots()
labels = np.arange(2, 10, 2)
labels = [str(x) for x in labels]
x = np.arange(len(labels))


embedding_dimensions_normal_recall = [embedding_dimension_2_normal_recall, embedding_di
                                        embedding_dimension_6_normal_recall, embedding_
embedding_dimensions_anomaly_recall = [embedding_dimension_2_anomaly_recall,  embedding
                                        embedding_dimension_6_anomaly_recall, embedding
embedding_dimensions_normal_recall = [round(x,4) for x in embedding_dimensions_normal_r
embedding_dimensions_anomaly_recall = [round(x,4) for x in embedding_dimensions_anomaly

embedding_dimensions_normal_bar = ax.bar(x - 0.2, embedding_dimensions_normal_recall, 0
                                          label='Embedding Dimension - Normal Recall',
                                          edgecolor='#203864', hatch='/')
embedding_dimensions_anomaly_bar = ax.bar(x + 0.2, embedding_dimensions_anomaly_recall,
                                          label='Embedding Dimension - Anomaly Recall',
                                          edgecolor='#843C0C', hatch='x')

ax.set_ylabel('Recall Percentage')
ax.set_xlabel('Embedding Dimension')
ax.set_title('Recall Percentage by Embedding Dimension')
ax.set_xticks(x, labels)
ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.1), fancybox=True, shadow=True, n
ax.bar_label(embedding_dimensions_normal_bar, padding=3)
ax.bar_label(embedding_dimensions_anomaly_bar, padding=3)


plt.show()
```



d) Briefly explain the trend you see in the *Recall* in part (c) above.

When the threshold is fixed, as the embedding dimension is increased, normal recall increases. As the embedding dimension is increased, loss decreases; therefore, records in the past step that surpassed the threshold, may not break the threshold in the current step since the loss is reduced.