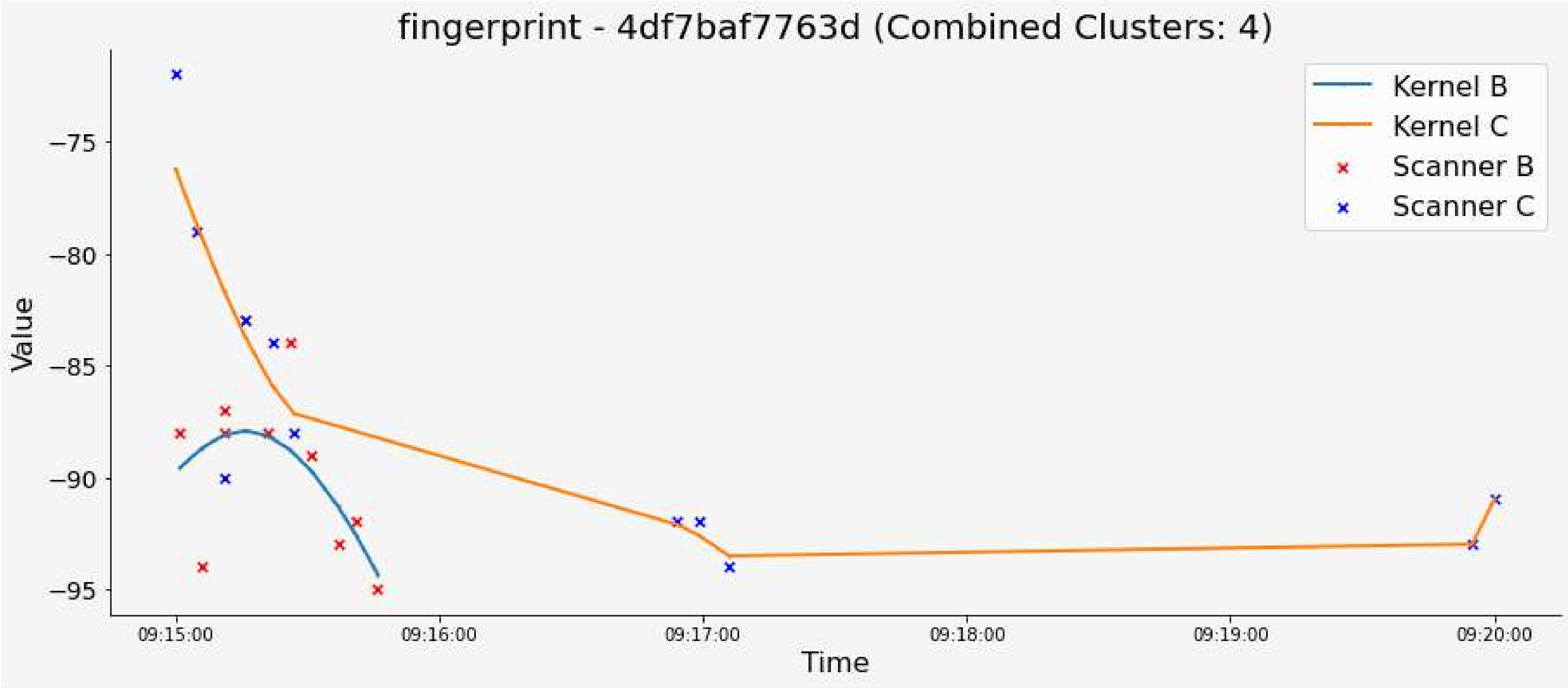
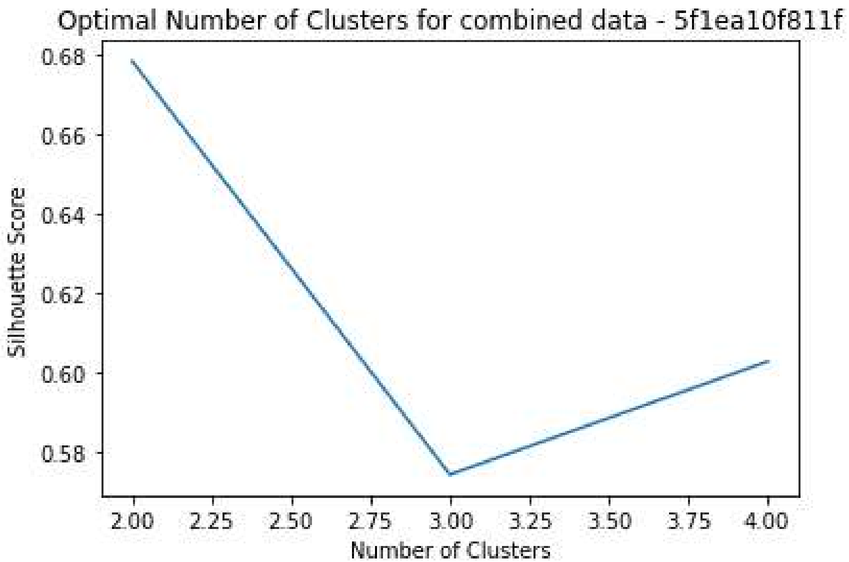
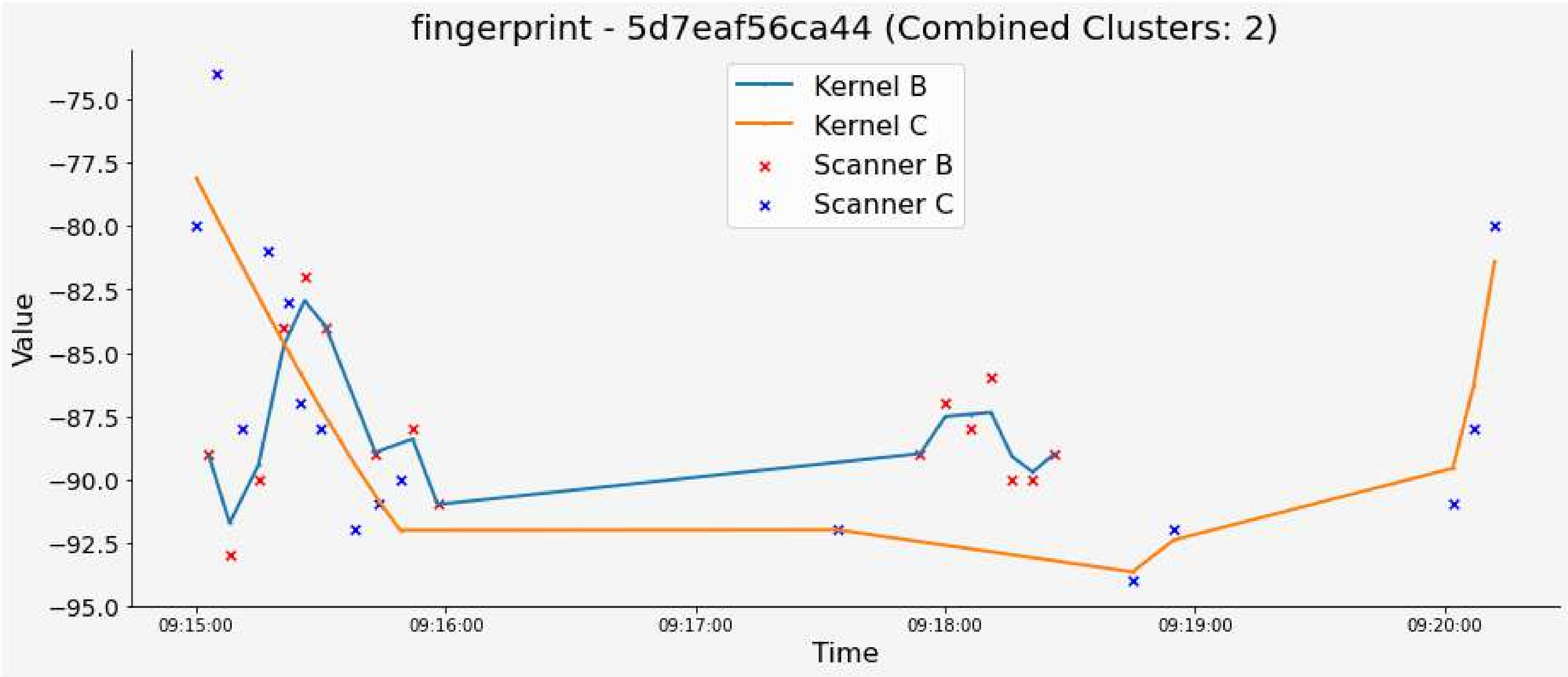
In [10]:
```
class TreeStructure(nn.Module):
    def __init__(self, middle_index, item_index,layer_top_emb,layer_bottom_emb,first_train):
        super(TreeStructure, self).__init__()

        # Parameters
        self.first_train = first_train
        self.ntokens = 72582#the number of ouput.(72582)
        self.nhid = 512#dimension: the same length of customer dimension.(512)

        self.ntokens_per_class = 20#how many children one intermidiate node.(20)

        self.nclasses = int(np.ceil(self.ntokens * 1. / self.ntokens_per_class))#intermidiate nodes.(3630)
        self.ntokens_actual = self.nclasses * self.ntokens_per_class#72600
        if self.first_train:
            self.layer_top_emb = nn.Parameter(torch.FloatTensor(self.nclasses,self.nhid), requires_grad=True)
            self.layer_bottom_emb = nn.Parameter(torch.FloatTensor(self.ntokens_actual, self.nhid), requires_grad=True)
            self.init_weights()
            #for K-means to cluster the embedding.(Initialization)
            self.middle_index = np.arange(self.nclasses).tolist()
            self.item_index = np.arange(self.ntokens_actual).tolist()
        else:
            #(Inherit from the previous K-means clustering)
            self.middle_index = middle_index.tolist()
            self.item_index = item_index.tolist()
            self.layer_top_emb = nn.Parameter(layer_top_emb, requires_grad=True)
            self.layer_bottom_emb = nn.Parameter(layer_bottom_emb, requires_grad=True)


    def init_weights(self):

        initrange = 0.1
        self.layer_top_emb.data.uniform_(-initrange, initrange)
        self.layer_bottom_emb.data.uniform_(-initrange, initrange)
    def forward(self, purchase_hist_npos):
        #leaf index
        hist = purchase_hist_npos

        #nonleaf index
        parent_index = (hist/ self.ntokens_per_class).long()#the position after clustering

        #leaf embedding
        positive_leaf_emb = self.layer_bottom_emb[hist]#positive 1###[256, 512]
        negative_leaf_sample = torch.LongTensor(np.random.choice(72600, positive_leaf_emb.shape[0]))###[256]
        negative_leaf_emb = self.layer_bottom_emb[negative_leaf_sample]#negative 1###[256, 512]
        #nonleaf embedding
        positive_nonleaf_emb = self.layer_top_emb[parent_index]#positive 2###[256, 512]
        negative_nonleaf_sample = torch.LongTensor(np.random.choice(3630, positive_leaf_emb.shape[0]))
        negative_nonleaf_emb = self.layer_top_emb[negative_nonleaf_sample]#negative 2


        return [positive_leaf_emb,negative_leaf_emb,positive_nonleaf_emb,negative_nonleaf_emb]
```

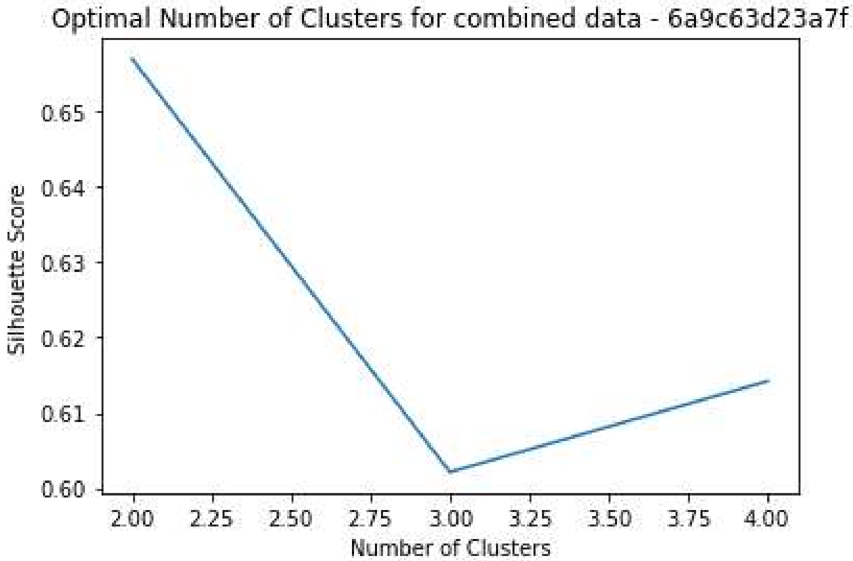
In [11]:
```
class HMModel(nn.Module):
    def __init__(self, article_shape,first_train,middle_index, item_index,layer_top_emb,layer_bottom_emb,pre_emb):
        super(HMModel, self).__init__()

        self.first_train = first_train
        if self.first_train:
            self.article_emb = torch.nn.Embedding(article_shape[0], embedding_dim=article_shape[1])
            middle_index = torch.ones(1)
            item_index = torch.ones(1)
            layer_top_emb = torch.ones(1)
            layer_bottom_emb = torch.ones(1)
        else:
            self.article_emb = torch.nn.Embedding.from_pretrained(torch.from_numpy(pre_emb).float())
            self.middle_index = middle_index
            self.item_index = item_index
            self.layer_top_emb = layer_top_emb
            self.layer_bottom_emb = layer_bottom_emb

        self.Tree = TreeStructure(middle_index, item_index,layer_top_emb,layer_bottom_emb,first_train=self.first_train)
    def forward(self, inputs):
        article_hist, week_hist, purchase_hist_npos = inputs[0], inputs[1], inputs[2]
        x = self.article_emb(article_hist)
        x = F.normalize(x, dim=2)###[256, 16, 512]

        x, indices = x.max(axis=1)##customer_emb[256,512]

        customer_emb = x
        global is_test

        if is_test:

            return customer_emb

        #print('0',purchase_hist_item,purchase_hist_item.shape)

        [p1,n1,p2,n2] = self.Tree(purchase_hist_npos)#get four logits for 2 positive and 2 negative samples

        p1_dot = torch.mul(x,p1).sum(dim=1).unsqueeze(0)
        n1_dot = torch.mul(x,n1).sum(dim=1).unsqueeze(0)
        p2_dot = torch.mul(x,p2).sum(dim=1).unsqueeze(0)
        n2_dot = torch.mul(x,n2).sum(dim=1).unsqueeze(0)


        logits = torch.cat((p1_dot,n1_dot,p2_dot,n2_dot),0).T

        return logits
middle_index = torch.ones(1)
item_index = torch.ones(1)
layer_top_emb = torch.ones(1)
layer_bottom_emb = torch.ones(1)
first_train = True
global first
global is_test
is_test = False
first = True
article_emb = torch.ones(1)
```


fingerprint - 4df7baf7763d (Combined Clusters: 4)
Kernel B
Kernel C
Scanner B
Scanner C
Value
−75
−80
−85
−90
−95
09:15:00
09:16:00
09:17:00
09:18:00
09:19:00
09:20:00
Time


Optimal Number of Clusters for combined data - 5f1ea10f811f
Silhouette Score
0.68
0.66
0.64
0.62
0.60
0.58
2.00
2.25
2.50
2.75
3.00
3.25
3.50
3.75
4.00
Number of Clusters

Number of Clusters





```python
#New data frame based on maximum of rssi by fingerprint df_1
idx=df_1.groupby('fingerprint')['rssi'].idxmax()
df_max1=df_1.loc[idx, ['fingerprint', 'rssi', 'timestamp']]

#New data frame based on maximum of rssi by fingerprint df_2
idx=df_2.groupby('fingerprint')['rssi'].idxmax()
df_max2=df_2.loc[idx, ['fingerprint', 'rssi', 'timestamp']]



# Create the time column for df_max1
df_max1['time'] = pd.to_datetime(df_max1['timestamp'], unit='s')
df_max1['time'] = df_max1['time'].dt.strftime('%H:%M:%S')
df_max1['time'] = df_max1['time'] - datetime.timedelta(hours=4)
df_max1['time'] = df_max1.time.astype(str).str.replace('0 days ', '')
df_max1['time'] = pd.to_datetime(df_max1['time']).dt.strftime('%H:%M:%S')



# Create the time column for df_max2
df_max2['time'] = pd.to_datetime(df_max2['timestamp'], unit='s')
df_max2['time'] = df_max2['time'].dt.strftime('%H:%M:%S')
df_max2['time'] = df_max2['time'] - datetime.timedelta(hours=4)
df_max2['time'] = df_max2.time.astype(str).str.replace('0 days ', '')
df_max2['time'] = pd.to_datetime(df_max2['time']).dt.strftime('%H:%M:%S')
```

```python
# Create the time column for df_1
df_1['time'] = pd.to_datetime(df_1['timestamp'], unit='s')
df_1['time'] = df_1['time'].dt.strftime('%H:%M:%S')
df_1['time'] = df_1['time'] - datetime.timedelta(hours=4)
df_1['time'] = df_1.time.astype(str).str.replace('0 days ', '')
df_1['time'] = pd.to_datetime(df_1['time']).dt.strftime('%H:%M:%S')

# Create the mode column : it reflects the times the signals captured the object moving
df_1['mode'] = df_1.groupby('fingerprint', sort=False).cumcount() + 1
```

Baseline model

In [ ]:
```
#import cross validation score
from sklearn.model_selection import cross_val_score

#import Naive Bayes Classifier
from sklearn.naive_bayes import GaussianNB

#create classifier object
nb = GaussianNB()

#run cv for NB classifier
from sklearn.metrics import classification_report

nb_accuracy = cross_val_score(nb,X_train,y_train.values.ravel(), cv=5, scoring ='accuracy')


print('nb_accuracy: ' +str(nb_accuracy))

print('nb_accuracy_avg: ' + str(nb_accuracy.mean()))
```

In [ ]:
```
#Let's now experiment with a few different basic models

## Logistic Regression
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression(random_state=32)

lr_accuracy = cross_val_score(lr,X_train,y_train.values.ravel(), cv=5, scoring ='accuracy')
lr_f1 = cross_val_score(lr,X_train,y_train.values.ravel(), cv=5, scoring ='f1')

print('lr_accuracy: ' +str(lr_accuracy))

print('lr_accuracy_avg: ' + str(lr_accuracy.mean()))
```