feature_engineering_optimized

March 19, 2021

1 Case Study 1

Predicting Central Neuropathic Pain (CNP) in people with Spinal Cord Injury (SCI) from Electroencephalogram (EEG) data.

- CNP is pain in response to non-painful stimuli, episodic (electric shock), "pins and needles", numbness
- There is currently no treatment, only prevention
- Preventative medications have strong side-effects
- Predicting whether a patient is likely to develop pain is useful for selective treatment

Task Your task is to devise a feature engineering strategy which, in combination with a classifier of your choice, optizimes prediction accuracy.

Data The data is preprocessed brain EEG data from SCI patients recorded while resting with eyes closed (EC) and eyes opened (EO). * 48 electrodes recording electrical activity of the brain at 250 Hz * 2 classes: subject will / will not develop neuropathic pain within 6 months * 18 subjects: 10 developed pain and 8 didn't develop pain * the data has already undergone some preprocessing * Signal denoising and normalization * Temporal segmentation * Frequency band power estimation * Normalization with respect to total band power * Features include normalized alpha, beta, theta band power while eyes closed, eyes opened, and taking the ratio of eo/ec. * the data is provided in a single table ('data.csv') consisting of * 180 rows (18 subjects x 10 repetitions), each containing * 432 columns (9 features x 48 electrodes) * rows are in subject major order, i.e. rows 0-9 are all samples from subject 0, rows 10-19 all samples from subject 1, etc. * columns are in feature_type major order, i.e. columns 0-47 are alpha band power, eyes closed, electrodes 0-48 * feature identifiers for all columns are stored in 'feature_names.csv' * 'labels.csv' defines the corresponding class (0 or 1) to each row in data.csv

Objective Measure Leave one subject out cross-validation accuracy, sensitivity and specificity.

Report Report on your feature engineering pipeline, the classifier used to evaluate performance, and the performance as mean and standard deviation of accuracy, sensitivity and specificity across folds. Give evidence for why your strategy is better than others.

[1]:
```
#Import all libraries needed at first for cleaner and clearer code
import csv
import numpy as np
import pandas as pd
```

```
import matplotlib.pyplot as plt
from IPython.display import display
import warnings
import seaborn as sns
warnings.filterwarnings('ignore')
from copy import deepcopy
import time
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LassoCV, LogisticRegression
from sklearn.feature_selection import SelectFromModel, SelectKBest,␣
↪mutual_info_classif, f_classif
from sklearn.feature_selection import VarianceThreshold, RFECV, RFE
from sklearn.pipeline import Pipeline
from sklearn.model_selection import LeaveOneGroupOut, cross_val_predict,␣
↪cross_val_score, cross_validate
from sklearn.svm import SVC, LinearSVC
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve, auc
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SequentialFeatureSelector as SFS
```

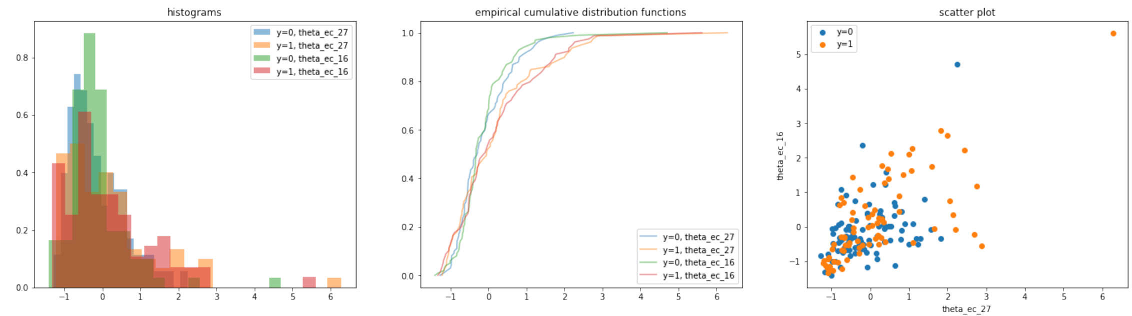
[2]:
```
# load data
# rows in X are subject major order, i.e. rows 0-9 are all samples from subject␣
↪0, rows 10-19 all samples from subject 1, etc.
# columns in X are in feature_type major order, i.e. columns 0-47 are alpha␣
↪band power, eyes closed, electrodes 0-48
# feature identifiers for all columns in X are stored in feature_names.csv
X = np.loadtxt('data.csv', delimiter=',')
y = np.loadtxt('labels.csv', delimiter=',')
with open('feature_names.csv') as f:
    csvreader = csv.reader(f, delimiter=',')
    feature_names = [row for row in csvreader][0]
```

[3]:
```
#Normalize feature data
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

[4]:
```
# plotting data in 2D with axes sampled
# a) at random
# b) from same electrode
# c) from same feature type
num_features = 9
num_electrodes = 48

# a) indices drawn at random
i0, i1 = np.random.randint(0, X.shape[1], size=2)
```

```
# b) same electrode, different feature (uncomment lines below)
#f0, f1 = np.random.randint(0, num_features, size=2)
#e = np.random.randint(0, num_electrodes)
#i0, i1 = f0*num_electrodes + e, f1*num_electrodes + e

# b) same feature, different electrode (uncomment lines below)
#f = np.random.randint(0, num_features)
#e0, e1 = np.random.randint(0, num_electrodes, size=2)
#i0, i1 = f*num_electrodes + e0, f*num_electrodes + e1

fig, axes = plt.subplots(1, 3, figsize=(24, 6))
colors = ['blue', 'red']

# select features i0, i1 and separate by class
X00, X01 = X[y==0][:,i0], X[y==1][:,i0]
X10, X11 = X[y==0][:,i1], X[y==1][:,i1]
# plot cumulative distribution of feature i0 separate for each class
axes[0].hist(X00, bins=20, label='y=0, '+ feature_names[i0], density=True,␣
↪alpha=0.5)
axes[0].hist(X01, bins=20, label='y=1, '+ feature_names[i0], density=True,␣
↪alpha=0.5)
axes[0].hist(X10, bins=20, label='y=0, '+ feature_names[i1], density=True,␣
↪alpha=0.5)
axes[0].hist(X11, bins=20, label='y=1, '+ feature_names[i1], density=True,␣
↪alpha=0.5)
axes[0].set_title('histograms')
axes[0].legend()
axes[1].plot(np.sort(X00), np.linspace(0,1,X00.shape[0]), label='y=0, '+␣
↪feature_names[i0], alpha=0.5)
axes[1].plot(np.sort(X01), np.linspace(0,1,X01.shape[0]), label='y=1, '+␣
↪feature_names[i0], alpha=0.5)
axes[1].plot(np.sort(X10), np.linspace(0,1,X10.shape[0]), label='y=0, '+␣
↪feature_names[i1], alpha=0.5)
axes[1].plot(np.sort(X11), np.linspace(0,1,X11.shape[0]), label='y=1, '+␣
↪feature_names[i1], alpha=0.5)
axes[1].set_title('empirical cumulative distribution functions')
axes[1].legend()
axes[2].scatter(X00, X10, label='y=0')
axes[2].scatter(X01, X11, label='y=1')
axes[2].set_xlabel(feature_names[i0])
axes[2].set_ylabel(feature_names[i1])
axes[2].set_title('scatter plot')
axes[2].legend()
```

```
[4]: <matplotlib.legend.Legend at 0x7fda56221160>
```



Step 1: Data Analysis and exploration

[5]:
```
# Extract data as dataframe to explore later on
features = pd.DataFrame(data=X, columns=feature_names)
```

[6]:
```
#Check for NaN values:
features.isnull().values.any()
```

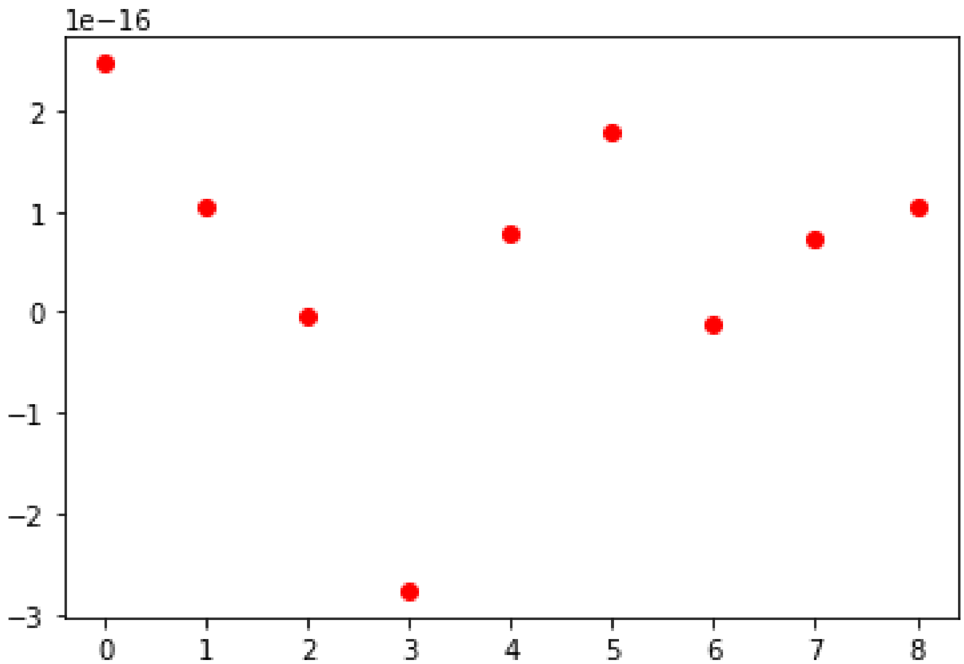
[6]: False

Combine features based on waves:

[7]:
```
#Alpha, beta and theta waves for eyes open
alpha_eo = features.iloc[:,0:48]
beta_eo = features.iloc[:,48:48*2]
theta_eo = features.iloc[:,48*2:48*3]
#Alpha, beta and theta waves for eyes closed
alpha_ec = features.iloc[:,48*3:48*4]
beta_ec = features.iloc[:,48*4:48*5]
theta_ec = features.iloc[:,48*5:48*6]
#Ratio for alpha beta and theta eyes open/ eyes closed
alpha_r = features.iloc[:,48*6:48*7]
beta_r = features.iloc[:,48*7:48*8]
theta_r = features.iloc[:,48*8:48*9]
#Add all features in a list for ease of manipulation
independent_variables =␣
↪[alpha_eo,alpha_ec,alpha_r,beta_eo,beta_ec,beta_r,theta_eo,theta_ec,theta_r]
```

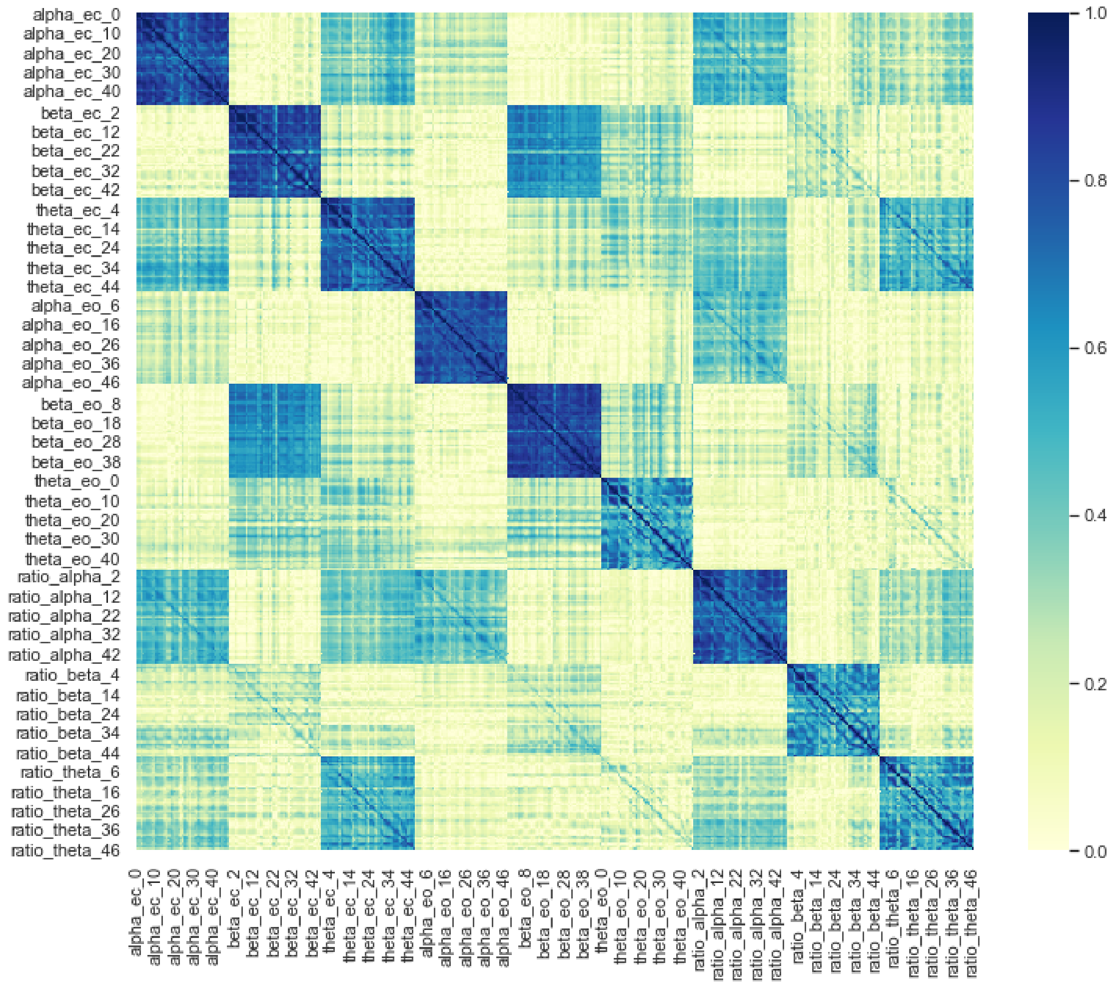
[8]:
```
#Get the means of each feature and plot them
means = []
for feature in independent_variables:
    means.append(np.mean(feature.values))
plt.scatter(range(0,len(independent_variables)),means,c="r")
```

[8]: <matplotlib.collections.PathCollection at 0x7fda5677f7f0>



[9]:
```
#Calculate correlation matrix to check the values of between 0 and 1 of all␣
↪features, it would be a good idea
#to eliminate features that are highly correlated
corr_matrix = features.corr(method = "spearman").abs()
```

[10]:
```
# Draw the heatmap on some examples
sns.set(font_scale = 1.0)
f, ax = plt.subplots(figsize=(11, 9))
sns.heatmap(corr_matrix, cmap= "YlGnBu", square=True, ax = ax)
f.tight_layout()
```

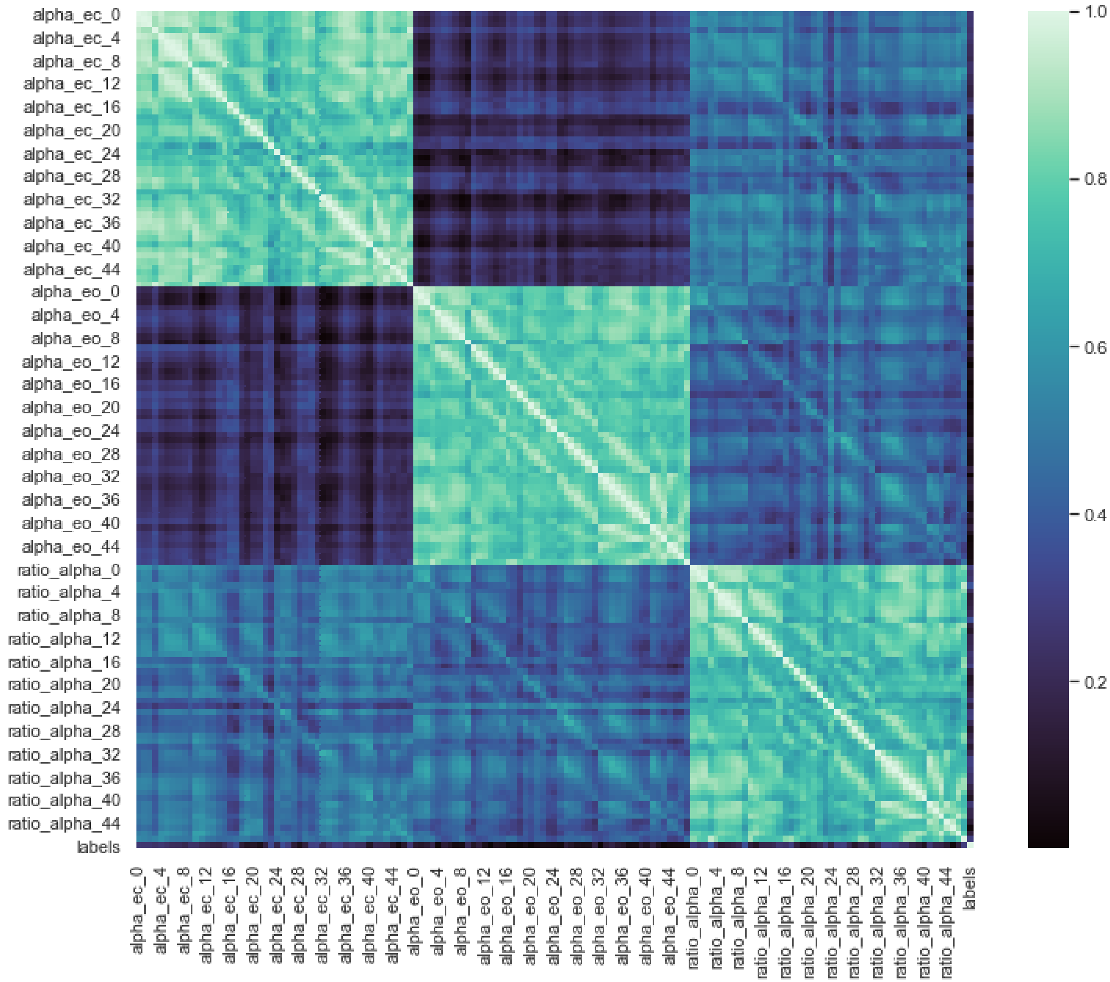



```
[11]: alpha_waves = independent_variables[0].join(independent_variables[1]).join(
          independent_variables[2]).join(
          pd.DataFrame(y,columns=["labels"]))

      beta_waves = independent_variables[3].join(independent_variables[4]).join(
          independent_variables[5]).join(
          pd.DataFrame(y,columns=["labels"]))

      theta_waves = independent_variables[6].join(independent_variables[7]).join(
          independent_variables[8]).join(
          pd.DataFrame(y,columns=["labels"]))
```

ALPHA WAVES

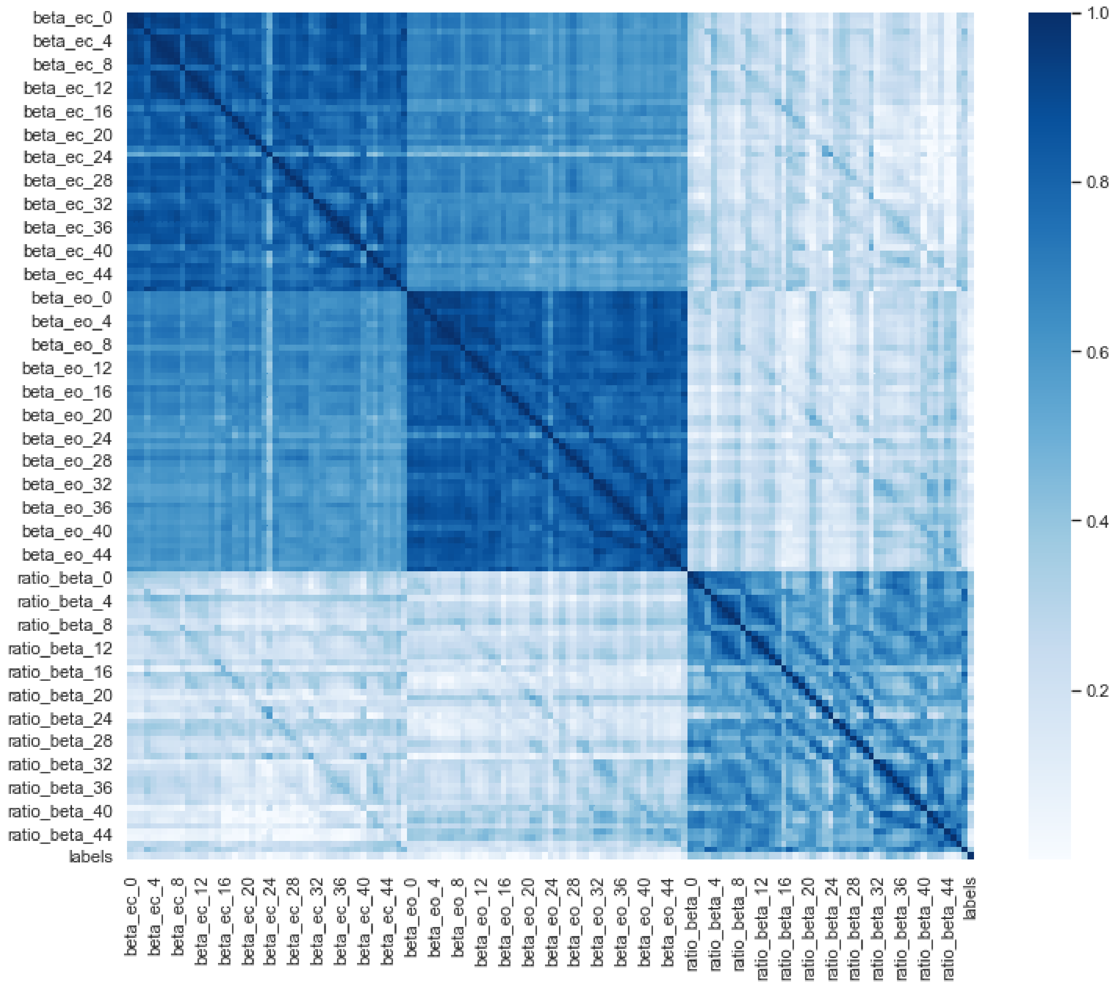
```
[12]: alpha_corr = alpha_waves.corr(method="spearman").abs()
      # Draw the heatmap on alpha waves
      sns.set(font_scale = 1.0)
```

```python
f, ax = plt.subplots(figsize=(11, 9))
sns.heatmap(alpha_corr, square=True, ax = ax, cmap="mako")
f.tight_layout()
```

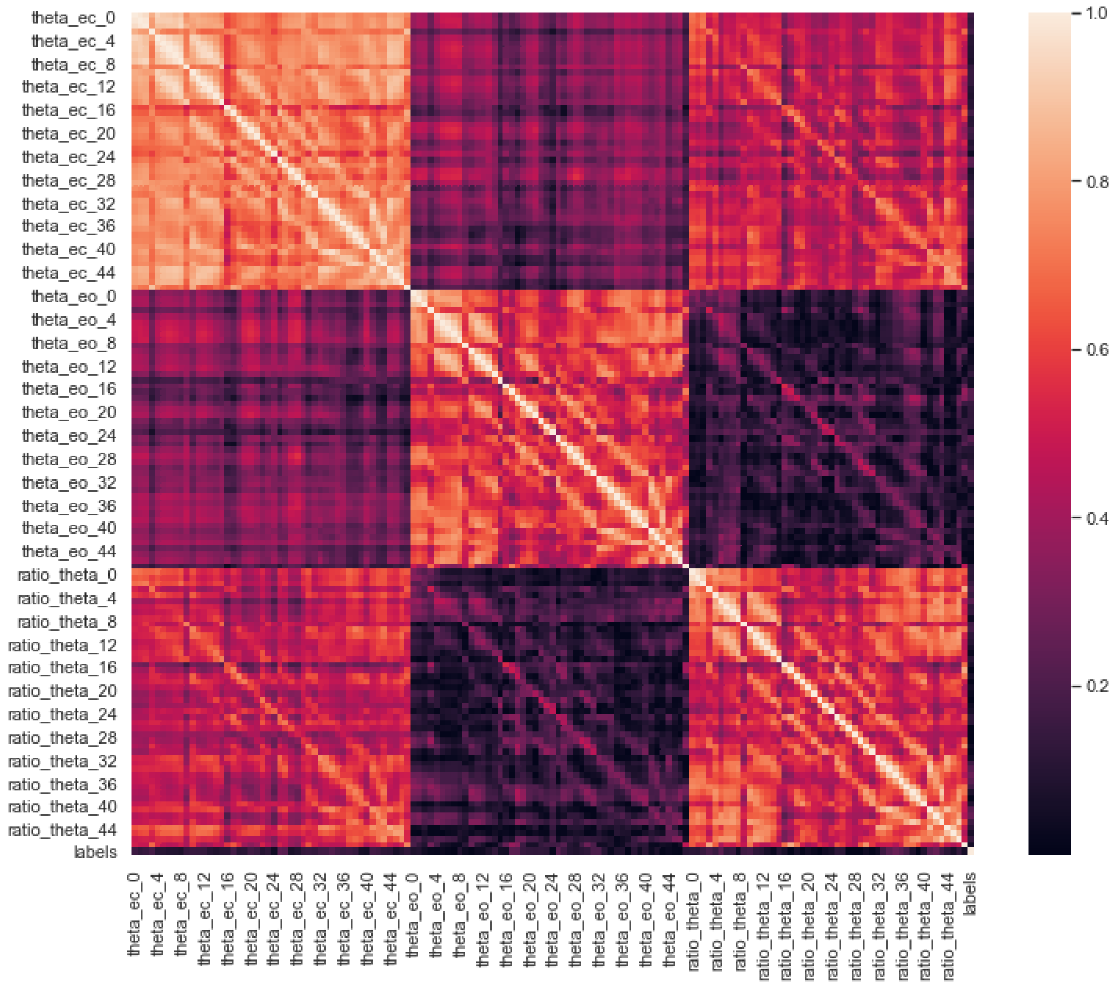


BETA WAVES CORRELATION

[13]:
```python
beta_corr = beta_waves.corr(method="spearman").abs()
# Draw the heatmap on beta waves
sns.set(font_scale = 1.0)
f, ax = plt.subplots(figsize=(11, 9))
sns.heatmap(beta_corr, square=True, ax = ax, cmap="Blues")
f.tight_layout()
```



[14]:
```
theta_corr = theta_waves.corr(method="spearman").abs()
# Draw the heatmap on theta waves
sns.set(font_scale = 1.0)
f, ax = plt.subplots(figsize=(11, 9))
sns.heatmap(theta_corr, square=True, ax = ax)
f.tight_layout()
```



From the correlation maps above, we can see that some features within the waves are highly correlated together, hence working as some kind of duplication of information.

Step 2: Feature engineering + Result Testing In this step, we will test some feature engineering methods to reduce feature dimensions in order to achieve higher performance.

[15]:
```
#Since the experiments needs one subject out, we will group every 9 rows as a subject

groups = np.zeros(180).astype(int)

counter = 0
subject = 0

while(counter < 180):
    if(counter%10==0 and counter !=0):
        subject+=1
```

```
    groups[counter] = subject
    counter+=1
```

[74]:
```
#This dataframe will hold the different results from the different methods used
results_final = pd.DataFrame()
```

[75]:
```
#This method will be used for ease of extracting the different metrics from the␣
↪confusion matrix
def get_scores(cm, y_preds):

    TP = cm[0][0]
    FP = cm[0][1]
    FN = cm[1][0]
    TN = cm[1][1]

    sp = TN/(TN + FP)
    se = TP/(TP + FN)
    acc = (TP + TN)/(TP+FP+FN+TN)
    pr = TP/(TP+FP)
    f1 = 2*pr*se/(pr+se)
    f, t, th = roc_curve(y, y_preds)

    auc_score = auc(f,t)

    return [sp, se, acc, pr, f1, auc_score]
```

[70]:
```
score_names = ['Specificity','Sensitivity','Accuracy','Precision', 'F1␣
↪Score','AUC Score', 'Execution Time (s)','Number of Features']
```

Part 0: Baseline

[76]:
```
y[y == 0] = -1 #Perform for SVM output of {-1,1}
```

[84]:
```
#Baseline Code:
start_time = time.time()

model = svc = LinearSVC(penalty="l1",dual=False,random_state=42)

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X, y, groups)

y_preds_0 = cross_val_predict(model, X, y, cv=cv, n_jobs=-1)

cm_0 = confusion_matrix(y, y_preds_0)

scores_base = get_scores(cm_0, y_preds_0)
```

```
execution_time = time.time() - start_time
```

[85]:
```
#Add results of baseline to aggregating table
scores_base.append(execution_time)
scores_base.append(int(X.shape[1]))
results_final["Baseline"] = scores_base
results_final.index = score_names
```

[86]:
```
#We can see that the baseline method has room for improvement.
results_final
```

[86]:

	Baseline
Specificity	0.880952
Sensitivity	0.937500
Accuracy	0.911111
Precision	0.900000
F1 Score	0.918367
AUC Score	0.912500
Execution Time (s)	0.559579
Number of Features	432.000000

Part 1: Filtering Methods

Method 1: Correlation Filter

[87]:
```
#Drop features with correlation > 95%
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k = 1).astype(np.
 ↪bool))
to_drop = [column for column in upper.columns if any(upper[column] > 0.95)]
reduced_features = features.drop(to_drop, axis = 1)
reduced_features.shape
```

[87]: (180, 350)

[88]:
```
#Remove quasi-constant features with no variance => no information to give
qconstant_filter = VarianceThreshold(threshold=0.01)
qconstant_filter.fit(reduced_features)
```

[88]: VarianceThreshold(threshold=0.01)

[89]:
```
#By the result above, we see that there are 14 quasi constant feature ! which␣
 ↪will allow us to further reduce the dims
reduced_features = qconstant_filter.transform(reduced_features)
#New reduced dimensions of features:
print(f"{reduced_features.shape[1]} features")
```

```
350 features
```

[91]:
```
reduced_features_df = pd.DataFrame(data=reduced_features)
```

[92]:
```
reduced_features_T = reduced_features.T
reduced_features_T = pd.DataFrame(data=reduced_features_T)
```

[93]:
```
print(reduced_features_T.duplicated().sum())
```

```
0
```

[94]:
```
#No duplicates were found. Good !
X_corr = reduced_features
print(f"{X_corr.shape[1]} features")
```

```
350 features
```

[95]:
```
start_time = time.time()

model = svc = LinearSVC(penalty="l1",dual=False,random_state=42)

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X_corr, y, groups)

y_preds_1 = cross_val_predict(model, X_corr, y, cv=cv, n_jobs=-1)

cm_1 = confusion_matrix(y, y_preds_1)

scores_corr = get_scores(cm_1, y_preds_1)

execution_time = time.time() - start_time
```

[96]:
```
#Add results to aggregating table
scores_corr.append(execution_time)
scores_corr.append(X_corr.shape[1])
results_final["High Correlation Filter"] = scores_corr
```

[97]:
```
#We can see that this filter degrades the performance. So we will count it out.
results_final
```

[97]:

	Baseline	High Correlation Filter
Specificity	0.880952	0.914634
Sensitivity	0.937500	0.948980
Accuracy	0.911111	0.933333
Precision	0.900000	0.930000
F1 Score	0.918367	0.939394
AUC Score	0.912500	0.933750
Execution Time (s)	0.559579	0.623599

```
Number of Features  432.000000              350.000000
```

Method 2: Mutual Information Filter

[102]:

```
#After applying the high correlation filter, we will use the Mutual Information␣
 ↪to select the k best features
#We will use GridSearch to find the best k to use

start_time = time.time()

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X, y, groups)

svc = LinearSVC(penalty="l1",dual=False,random_state=42)

N_FEATURES_OPTIONS = range(1,features.shape[1],24)

mutual_info = SelectKBest(score_func=mutual_info_classif)

param = {'mutual_reduce_dim': [mutual_info],'mutual_reduce_dim__k':␣
 ↪N_FEATURES_OPTIONS}

pipe = Pipeline(steps=[('mutual_reduce_dim','passthrough'), ('classify', svc)])

grid = GridSearchCV(pipe, n_jobs=-1, param_grid=param, cv=cv)

grid.fit(X,y)

#After getting the best k number of features based on accuracy score, we will␣
 ↪use it and get all metrics.
best_k = grid.best_params_["mutual_reduce_dim__k"]
mutual_info_best = SelectKBest(score_func=mutual_info_classif,k=best_k)
mutual_info_best.fit(X,y)
X_mutual = mutual_info_best.transform(X)
print(f"{X_mutual.shape[1]} features")

#Test the mutual info method
model = svc = LinearSVC(penalty="l1",dual=False,random_state=42)

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X_mutual, y, groups)

y_preds_2 = cross_val_predict(model, X_mutual, y, cv=cv, n_jobs=-1)

cm_2 = confusion_matrix(y, y_preds_2)
```

```
scores_mutual = get_scores(cm_2, y_preds_2)
execution_time = time.time() - start_time
```

313 features

[104]:
```
#Add results to aggregating table
scores_mutual.append(execution_time)
scores_mutual.append(X_mutual.shape[1])
results_final["Mutual Information Filter"] = scores_mutual
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-104-dff8d01539e6> in <module>
      2 scores_mutual.append(execution_time)
      3 scores_mutual.append(X_mutual.shape[1])
----> 4 results_final["Mutual Information Filter"] = scores_mutual

~/opt/anaconda3/envs/venv/lib/python3.8/site-packages/pandas/core/frame.py in␣
↪__setitem__(self, key, value)
   3161         else:
   3162             # set column
-> 3163             self._set_item(key, value)
   3164 
   3165     def _setitem_slice(self, key: slice, value):

~/opt/anaconda3/envs/venv/lib/python3.8/site-packages/pandas/core/frame.py in␣
↪_set_item(self, key, value)
   3237         """
   3238         self._ensure_valid_index(value)
-> 3239         value = self._sanitize_column(key, value)
   3240         NDFrame._set_item(self, key, value)
   3241 

~/opt/anaconda3/envs/venv/lib/python3.8/site-packages/pandas/core/frame.py in␣
↪_sanitize_column(self, key, value, broadcast)
   3894 
   3895             # turn me into an ndarray
-> 3896             value = sanitize_index(value, self.index)
   3897             if not isinstance(value, (np.ndarray, Index)):
   3898                 if isinstance(value, list) and len(value) > 0:

~/opt/anaconda3/envs/venv/lib/python3.8/site-packages/pandas/core/internals/
↪construction.py in sanitize_index(data, index)
    749     """
    750     if len(data) != len(index):
```

```
--> 751         raise ValueError(
    752             "Length of values "
    753             f"({len(data)}) "

ValueError: Length of values (10) does not match length of index (8)
```

[105]:
```
results_final
```

[105]:
```
                      Baseline  High Correlation Filter  \
Specificity           0.880952                 0.914634
Sensitivity           0.937500                 0.948980
Accuracy              0.911111                 0.933333
Precision             0.900000                 0.930000
F1 Score              0.918367                 0.939394
AUC Score             0.912500                 0.933750
Execution Time (s)    0.559579                 0.623599
Number of Features  432.000000               350.000000

                    Mutual Information Filter
Specificity                          0.924051
Sensitivity                          0.930693
Accuracy                             0.927778
Precision                            0.940000
F1 Score                             0.935323
AUC Score                            0.926250
Execution Time (s)                 182.574039
Number of Features                 313.000000
```

Method 3: ANOVA Filter (f-Test)

[106]:
```
#After applying the high correlation filter, we will use the ANOVA (f-Test) to␣
↪select the k best features
#We will use GridSearch to find the best k to use

start_time = time.time()

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X, y, groups)

svc = LinearSVC(penalty="l1",dual=False,random_state=42)

N_FEATURES_OPTIONS = range(1,features.shape[1],24)

anova = SelectKBest(score_func=f_classif)
```

```
param = {'anova_reduce_dim': [anova], 'anova_reduce_dim__k': N_FEATURES_OPTIONS}

pipe = Pipeline(steps=[('anova_reduce_dim','passthrough'), ('classify', svc)])

grid = GridSearchCV(pipe, n_jobs=-1, param_grid=param, cv=cv)

grid.fit(X,y)

#After getting the best k number of features based on accuracy score, we will␣
↪use it and get all metrics.
best_k = grid.best_params_["anova_reduce_dim__k"]

anova_best = SelectKBest(score_func=f_classif,k=best_k)

anova_best.fit(X,y)

X_anova = anova_best.transform(X)

print(f"{X_anova.shape[1]} features")

#Test the mutual info method
model = LinearSVC(penalty="l1",dual=False,random_state=42)

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X_anova, y, groups)

y_preds_3 = cross_val_predict(model, X_anova, y, cv=cv, n_jobs=-1)

cm_3 = confusion_matrix(y, y_preds_3)

scores_anova = get_scores(cm_3, y_preds_3)

execution_time = time.time() - start_time
```

```
289 features
```

[107]:
```
#Add results to aggregating table
scores_anova.append(execution_time)
scores_anova.append(X_anova.shape[1])
results_final["Anova (f-Test) Filter"] = scores_anova
```

[108]:
```
results_final
```

[108]:
```
                            Baseline  High Correlation Filter  \
Specificity                 0.880952                 0.914634
Sensitivity                 0.937500                 0.948980
```

```
Accuracy               0.911111              0.933333
Precision              0.900000              0.930000
F1 Score               0.918367              0.939394
AUC Score              0.912500              0.933750
Execution Time (s)     0.559579              0.623599
Number of Features   432.000000            350.000000

                    Mutual Information Filter  Anova (f-Test) Filter
Specificity                          0.924051               0.925926
Sensitivity                          0.930693               0.949495
Accuracy                             0.927778               0.938889
Precision                            0.940000               0.940000
F1 Score                             0.935323               0.944724
AUC Score                            0.926250               0.938750
Execution Time (s)                 182.574039               6.948190
Number of Features                 313.000000             289.000000
```

Part 2: Wrapper Methods

Method 1: Recursive Elimination with Cross Validation

[109]:
```python
start_time = time.time()

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X, y, groups)

svc = LinearSVC(penalty="l1",dual=False,random_state=42)

rfecv = RFECV(estimator=svc, min_features_to_select=1, cv=cv,n_jobs=-1)

X_tmp = deepcopy(X)

rfecv.fit(X_tmp, y)

print("Optimal number of features : %d" % rfecv.n_features_)

rfe = RFE(svc, n_features_to_select=rfecv.n_features_)

pipeline = Pipeline(steps=[('s',rfe),('m',svc)])

cv = LOGO.split(X, y, groups)

y_preds_4 = cross_val_predict(pipeline, X, y, cv=cv, n_jobs=-1)

cm_4 = confusion_matrix(y, y_preds_4)

scores_rfecv = get_scores(cm_4, y_preds_4)
```

```python
#After that, we can start with our training and testing using different methods
↪for dimensionality reduction.
execution_time = time.time() - start_time
```

```
Optimal number of features : 23
```

[110]:
```python
scores_rfecv.append(execution_time)
scores_rfecv.append(rfecv.n_features_)
results_final["RFECV"] = scores_rfecv
```

[111]:
```python
results_final
```

[111]:
```
                      Baseline  High Correlation Filter  \
Specificity           0.880952                 0.914634
Sensitivity           0.937500                 0.948980
Accuracy              0.911111                 0.933333
Precision             0.900000                 0.930000
F1 Score              0.918367                 0.939394
AUC Score             0.912500                 0.933750
Execution Time (s)    0.559579                 0.623599
Number of Features  432.000000               350.000000

                    Mutual Information Filter  Anova (f-Test) Filter  \
Specificity                          0.924051               0.925926
Sensitivity                          0.930693               0.949495
Accuracy                             0.927778               0.938889
Precision                            0.940000               0.940000
F1 Score                             0.935323               0.944724
AUC Score                            0.926250               0.938750
Execution Time (s)                 182.574039               6.948190
Number of Features                 313.000000             289.000000

                         RFECV
Specificity           0.926829
Sensitivity           0.959184
Accuracy              0.944444
Precision             0.940000
F1 Score              0.949495
AUC Score             0.945000
Execution Time (s)  292.743974
Number of Features   23.000000
```

Part 2: Embedded Methods

Method 1: Ridge Regression

[112]:
```
start_time = time.time()

selection = SelectFromModel(LinearSVC(C=1, penalty='l2',␣
↪dual=False,random_state=42))

selection.fit(X, y)

# see the selected features.
selected_features = features.columns[(selection.get_support())]

cols = []

for f in selected_features:
    cols.append(f)

X_ridge = features[selected_features]

print(f"{X_ridge.shape[1]} features")

model = LinearSVC(penalty="l1",dual=False, random_state=42)

LOGO = LeaveOneGroupOut()

cv = LOGO.split(X_ridge, y, groups)

y_preds_6 = cross_val_predict(model, X_ridge, y, cv=cv, n_jobs=-1)

cm_6 = confusion_matrix(y, y_preds_6)

scores_ridge = get_scores(cm_6, y_preds_6)

execution_time = time.time() - start_time
```

```
183 features
```

[113]:
```
scores_ridge.append(execution_time)
scores_ridge.append(X_ridge.shape[1])
results_final["Ridge"] = scores_ridge
```

[114]:
```
results_final
```

[114]:
```
                   Baseline  High Correlation Filter  \
Specificity        0.880952                 0.914634
Sensitivity        0.937500                 0.948980
Accuracy           0.911111                 0.933333
Precision          0.900000                 0.930000
F1 Score           0.918367                 0.939394
```

```
AUC Score              0.912500                 0.933750
Execution Time (s)     0.559579                 0.623599
Number of Features   432.000000               350.000000

                    Mutual Information Filter  Anova (f-Test) Filter  \
Specificity                          0.924051               0.925926
Sensitivity                          0.930693               0.949495
Accuracy                             0.927778               0.938889
Precision                            0.940000               0.940000
F1 Score                             0.935323               0.944724
AUC Score                            0.926250               0.938750
Execution Time (s)                 182.574039               6.948190
Number of Features                 313.000000             289.000000

                         RFECV       Ridge
Specificity           0.926829    0.895349
Sensitivity           0.959184    0.968085
Accuracy              0.944444    0.933333
Precision             0.940000    0.910000
F1 Score              0.949495    0.938144
AUC Score             0.945000    0.936250
Execution Time (s)  292.743974    0.776297
Number of Features   23.000000  183.000000
```

Method 2: Lasso Regression

[115]:
```
start_time = time.time()

selection = SelectFromModel(LinearSVC(C=1, penalty='l1',dual =␣
 ↪False,random_state=42))

selection.fit(X, y)

# see the selected features.
selected_features = features.columns[(selection.get_support())]

cols = []

for f in selected_features:
    cols.append(f)

X_lasso = features[selected_features]

print(f"{X_lasso.shape[1]} features")

model = LinearSVC(penalty="l1",dual=False,random_state=42)
```

```python
LOGO = LeaveOneGroupOut()

cv = LOGO.split(X_lasso, y, groups)

y_preds_7 = cross_val_predict(model, X_lasso, y, cv=cv, n_jobs=-1)

cm_7 = confusion_matrix(y, y_preds_7)

scores_lasso = get_scores(cm_7, y_preds_7)

execution_time = time.time() - start_time
```

59 features

[116]:
```python
scores_lasso.append(execution_time)
scores_lasso.append(X_lasso.shape[1])
results_final["Lasso"] = scores_lasso
```

[117]:
```python
results_final
```

[117]:
```
                       Baseline  High Correlation Filter  \
Specificity            0.880952                 0.914634
Sensitivity            0.937500                 0.948980
Accuracy               0.911111                 0.933333
Precision              0.900000                 0.930000
F1 Score               0.918367                 0.939394
AUC Score              0.912500                 0.933750
Execution Time (s)     0.559579                 0.623599
Number of Features   432.000000               350.000000

                     Mutual Information Filter  Anova (f-Test) Filter  \
Specificity                           0.924051               0.925926
Sensitivity                           0.930693               0.949495
Accuracy                              0.927778               0.938889
Precision                             0.940000               0.940000
F1 Score                              0.935323               0.944724
AUC Score                             0.926250               0.938750
Execution Time (s)                  182.574039               6.948190
Number of Features                  313.000000             289.000000

                        RFECV     Ridge     Lasso
Specificity          0.926829  0.895349  0.975610
Sensitivity          0.959184  0.968085  1.000000
Accuracy             0.944444  0.933333  0.988889
Precision            0.940000  0.910000  0.980000
F1 Score             0.949495  0.938144  0.989899
AUC Score            0.945000  0.936250  0.990000
```

```
Execution Time (s)   292.743974    0.776297   0.374280
Number of Features    23.000000  183.000000  59.000000
```

[118]:
```
results_final.to_csv('Methods_Scores.csv')  #Uncomment if file does not exist
```

[119]:
```
auc_scores = results_final.loc["AUC Score",:]
legends = ["B M0","F M1","F M2","F M3","W M1","E M1"]
mean_auc = np.median(auc_scores)
colors = ['b', 'c', 'y', 'm', 'g','k','m']
x = np.linspace(0,5,6)
ax.set_facecolor('w')
m0 = plt.scatter(x[0],auc_scores[0],color = colors[0])
m1 = plt.scatter(x[1],auc_scores[1],color = colors[1])
m2 = plt.scatter(x[2],auc_scores[2],color = colors[2])
m3 = plt.scatter(x[3],auc_scores[3],color = colors[3])
m4 = plt.scatter(x[4],auc_scores[4],color = colors[4])
m5 = plt.scatter(x[3],auc_scores[3],color = colors[6],marker="*")
plt.legend((m0,m1,m2,m3,m4,m5),
           (legends),
           scatterpoints=1,
           loc='lower right',
           ncol=3,
           fontsize=8)
plt.hlines(mean_auc,min(x),max(x),'r')
plt.savefig('results_graph.png')
```