

af_classification

December 5, 2024

1 Atrial Fibrillation Classification

The goal of this exercise is to train different conventional classification models to discriminate between atrial fibrillation and normal sinus rhythm from a sequence of interbeat intervals. We use interbeat intervals extracted from the Long Term AF Database (<https://physionet.org/content/ltafdb/1.0.0/>).

We will train the following models on windows of interbeat intervals:

- Decision tree
- Support vector machine (SVM)
- Naive Bayes

The models will be trained on simple features derived from each window of interbeat intervals.

First, we import all the required packages, define global constants, and seed the random number generators to obtain reproducible results.

```
[1]: %matplotlib widget

import operator
import pathlib
import warnings
import IPython.display
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import sklearn.metrics
import sklearn.model_selection
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import Lasso
import seaborn as sns
```

```
DATA_FILE = pathlib.Path('../data/ltafdb_intervals.npz')
LOG_DIRECTORY = pathlib.Path('../logs/af_classification')
```

Then, we load the windows of interbeat intervals and the corresponding labels. We also load the record identifiers. They will help to avoid using intervals from the same record for both training and testing.

```
[2]: def load_data():
    with np.load(DATA_FILE) as data:
        intervals = data['intervals']
        labels = data['labels']
        identifiers = data['identifiers']
    return intervals, labels, identifiers

intervals, labels, identifiers = load_data()
targets = (labels == 'atrial_fibrillation').astype('float32')[:, None]
window_size = intervals.shape[1]

print(f'Number of windows: {intervals.shape[0]}')
print(f'Window size: {window_size}')
print(f'Window labels: {set(labels)})
```

Number of windows: 25064
Window size: 32
Window labels: {'atrial_fibrillation', 'normal_sinus_rhythm'}

Here are a few examples of windows of interbeat intervals.

```
[3]: def plot_interval_examples(intervals, targets, n_examples=3):
    normal_indices = np.random.choice(np.flatnonzero(targets == 0.0), n_examples, replace=False)
    af_indices = np.random.choice(np.flatnonzero(targets == 1.0), n_examples, replace=False)

    def plot_intervals(ax, index):
        ax.plot(np.cumsum(intervals[index]), intervals[index], '.-')
        ax.grid(True)

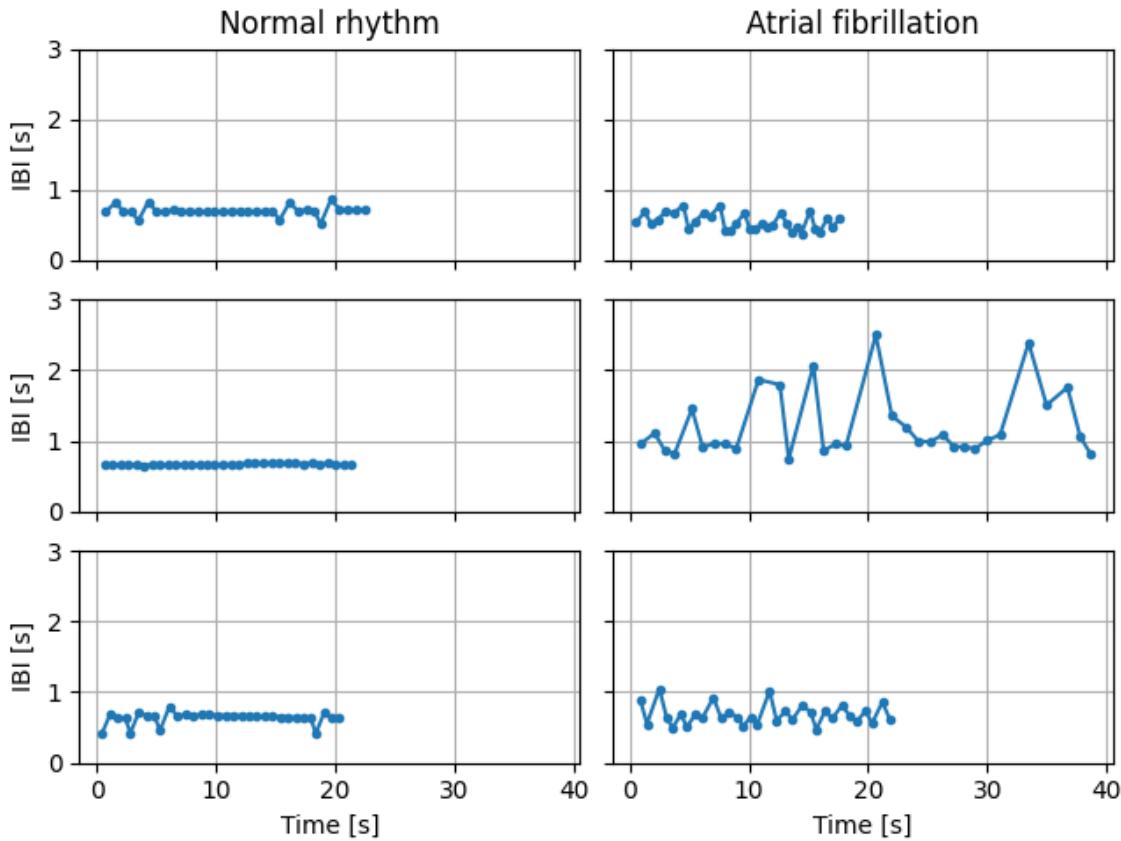
    fig, axes = plt.subplots(n_examples, 2, sharex='all', sharey='all', squeeze=False, constrained_layout=True)
    for i in range(n_examples):
        plot_intervals(axes[i, 0], normal_indices[i])
        plot_intervals(axes[i, 1], af_indices[i])
    plt.setp(axes, ylim=(0.0, 3.0))
    plt.setp(axes[-1, :], xlabel='Time [s]')
```

```

plt.setp(axes[:, 0], ylabel='IBI [s]')
axes[0, 0].set_title('Normal rhythm')
axes[0, 1].set_title('Atrial fibrillation')

plot_interval_examples(intervals, targets)

```



The next step is to split the dataset into subsets for training, validation, and testing stratified by labels.

```

[4]: def split_data(identifiers, intervals, targets):
    splitter = sklearn.model_selection.StratifiedGroupKFold(n_splits=5)
    indices = list(map(operator.itemgetter(1), splitter.split(intervals, u
    ↪targets, identifiers)))
    i_train = np.hstack(indices[:-2])
    i_val = indices[-2]
    i_test = indices[-1]

    assert not (set(identifiers[i_train]) & set(identifiers[i_val]))
    assert not (set(identifiers[i_train]) & set(identifiers[i_test]))
    assert not (set(identifiers[i_val]) & set(identifiers[i_test]))

```

```

    assert set(identifiers[i_train]) | set(identifiers[i_val]) | set(identifiers[i_test]) == set(identifiers)

    return i_train, i_val, i_test

i_train, i_val, i_test = split_data(identifiers, intervals, targets)

def build_summary(subsets, targets):
    data = []
    for subset, y in zip(subsets, targets):
        data.append({
            'subset': subset,
            'total_count': y.size,
            'normal_count': np.sum(y == 0.0),
            'af_count': np.sum(y == 1.0),
            'normal_proportion': np.mean(y == 0.0),
            'af_proportion': np.mean(y == 1.0),
        })
    return pd.DataFrame(data)

IPython.display.display(build_summary(['training', 'validation', 'testing'],
                                      (targets[i_train], targets[i_val], targets[i_test])))

```

	subset	total_count	normal_count	af_count	normal_proportion	af_proportion
0	training	15000	6919	8081	0.461267	0.538733
1	validation	4964	2365	2599	0.476430	0.523570
2	testing	5100	2311	2789	0.453137	0.546863

To better understand the dataset, we extract two features from each window of interbeat intervals: the mean and the standard deviation. We then plot these two features for the two classes.

```
[5]: f_mean = np.mean(intervals, axis=1)
f_std = np.std(intervals, axis=1)
f_min = np.min(intervals, axis=1)
f_max = np.max(intervals, axis=1)
f_median = np.median(intervals, axis=1)
features = pd.DataFrame({
    'mean': f_mean,
    'std': f_std,
    'min': f_min,
```

```

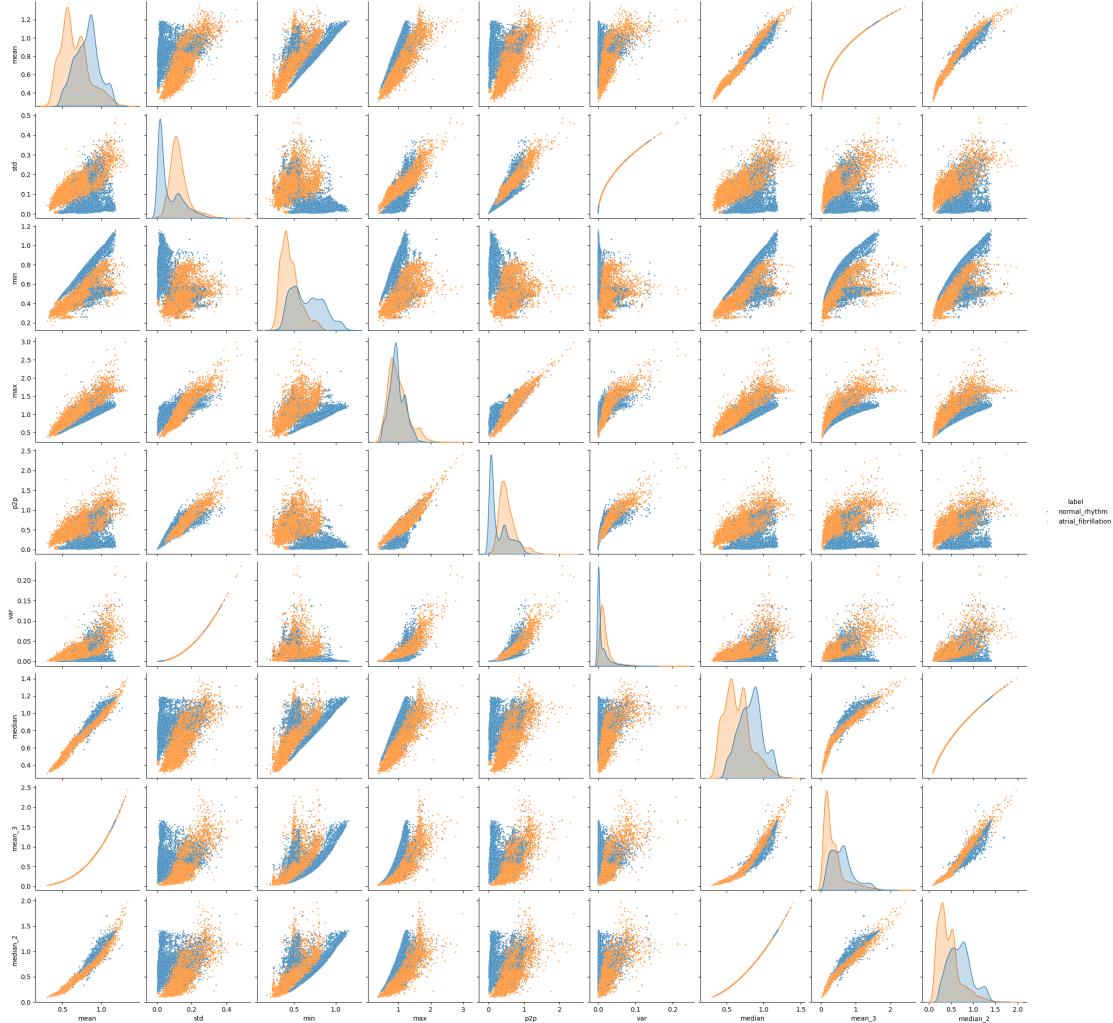
'max': f_max,
'p2p': f_max - f_min,
'var': np.power(f_std, 2),
'median': f_median,
'mean_3': np.power(f_mean, 3),
'median_2': np.power(f_median, 2),

})

def plot_features(f, y):
    data = f.copy()
    data['label'] = y.ravel()
    data['label'] = data['label'].map({0.0: 'normal_rhythm', 1.0: u
    ↵'atrial_fibrillation'})
    sns.pairplot(data, hue='label', plot_kws={'s': 4})

plot_features(features.iloc[i_train], targets[i_train])

```



It is also possible to select the most relevant features using various methods. Here, we define and implement three feature selection techniques: lasso, univariate, hybrid.

```
[6]: class FeatureSelector:

    def __init__(self, method, numbers):
        self.method = method.lower()
        self.numbers = numbers

    def apply(self, features, targets):
        features_names = [column for column in features.columns if column not_in ['reference']]
        features_selection = features.copy()
        features_selection.insert(0, 'reference', targets)
```

```

        features_selection = features_selection.dropna(axis=0, how='any', u
        ↪inplace=False)
        ranks = self.rank_features(features_selection[features_names],
                                    features_selection['reference'],
                                    self.method)

    def features_selection
        return self.select_features(ranks, self.numbers)

    @staticmethod
    def select_features(ranks, feature_num):
        ranks.sort_values(by='ranks', axis=0, ascending=False, inplace=True,
                          kind='quicksort', ignore_index=True)
        return ranks['feature_names'].iloc[: feature_num].tolist()

    @staticmethod
    def rank_features(features, reference, method):
        def univariate_selection(data, ref):
            selector = SelectKBest(f_regression, k="all")
            scores = selector.fit(data, ref).scores_
            return scores / np.nansum(scores)

        def lasso_selection(data, ref):
            alphas = np.arange(0.01, 0.3, 0.01)
            coefficients = np.empty([len(alphas), data.shape[1]])
            for row, alpha in enumerate(alphas):
                selector = SelectFromModel(Lasso(alpha=alpha), prefit=False)
                coefficients[row, :] = selector.fit(data, ref).estimator_.coef_
            coefficients = np.abs(coefficients)
            real_ranks = np.nansum(coefficients, axis=0)
            return real_ranks / np.nansum(real_ranks)

        if method == 'lasso':
            ranks = lasso_selection(features, reference)
        elif method == 'univariate':
            ranks = univariate_selection(features, reference)
        elif method == 'hybrid':
            rank_lasso = lasso_selection(features, reference)
            rank_univariate = univariate_selection(features, reference)
            rank_combined = rank_lasso + rank_univariate
            ranks = rank_combined / np.nansum(rank_combined)
        else:
            raise TypeError("Feature selection method is not supported")
        return pd.DataFrame({
            'feature_names': features.columns,
            'ranks': ranks,
        })

```

```

feature_selection_method = 'lasso'
feature_selection_numbers = 5
features_list = FeatureSelector(feature_selection_method,
    ↪feature_selection_numbers).apply(features.iloc[i_train], targets[i_train])
print(f"Selected features:{features_list}")

```

Selected features:['median_2', 'p2p', 'mean', 'std', 'min']

To classify atrial fibrillation and normal rhythm, we define the following models: Decision Tree, SVM, and Naive Bayes. To this end, we define a model builder class which provides a method to build the models.

[7]: `class ModelBuilder:`

```

def __init__(self, config):
    self.config = config['model']

def apply(self):
    return eval(f"self._build_{self.config['name']}()")

def _build_tree(self, max_depth=5):
    if 'max_depth' in self.config.keys():
        max_depth = self.config['max_depth']
    return make_pipeline(
        StandardScaler(),
        DecisionTreeClassifier(max_depth=max_depth))

def _build_svm(self, kernel='rbf', gamma='scale', regularization=1):
    if 'kernel' in self.config.keys():
        kernel = self.config['kernel']
    if 'gamma' in self.config.keys():
        gamma = self.config['gamma']
    if 'regularization' in self.config.keys():
        regularization = self.config['regularization']
    return make_pipeline(
        StandardScaler(),
        SVC(kernel=kernel, gamma=gamma, C=regularization,
             probability=True))

def _build_bayes(self, var_smoothing=1e-09):
    if 'var_smoothing' in self.config.keys():
        var_smoothing = self.config['var_smoothing']
    return make_pipeline(
        StandardScaler(),
        GaussianNB(var_smoothing=var_smoothing))

```

Now, we define a class for the training of the models.

```
[8]: class ModelTrainer:

    def __init__(self, config):
        self.config = config['feature']

    def apply(self, model, features, reference, i_train):
        features_list = self._get_features_list(list(features.columns))
        features_train = features.copy()
        features_train.insert(0, 'reference', reference)
        features_train = features_train.iloc[i_train].copy()
        features_train = features_train.dropna(axis=0, how='any', inplace=False,
                                              subset=features_list + [
                                                'reference'])
        return model.fit(
            features_train[features_list].values, features_train['reference'].values)

    def _get_features_list(self, current_features):
        if 'all' in self.config['list']:
            features_list = [feature for feature in current_features
                            if feature not in self.config['exclusion']] + [
                'reference']
        else:
            features_list = [feature for feature in
                            self.config['list']
                            if feature in current_features and feature not
                            in self.config['exclusion']] + ['reference']
        return features_list
```

We also define a class to apply the trained models on the test data.

```
[9]: class ModelTester:

    def __init__(self, config):
        self.config = config['feature']

    def apply(self, model, features):
        features_list = self._get_features_list(list(features.columns))
        inx = np.logical_not(
            np.sum(np.isnan(features[features_list]), 1).astype(bool))
        detections = np.zeros_like(inx)
        detections[:] = np.nan
        detections[inx] = model.predict(features[features_list].values[inx])
        return pd.DataFrame({'prediction': detections})

    def _get_features_list(self, current_features):
        if 'all' in self.config['list']:
```

```

        features_list = [feature for feature in current_features
                          if feature not in self.config['exclusion']] + [
            'reference']
    else:
        features_list = [feature for feature in self.config['list']
                          if feature in current_features and feature not
                          in self.config['exclusion']] + ['reference']
    return features_list

```

Inorder to evaluate the results of the models, we define an Evaluator class as follows:

```
[10]: class Evaluator:

    def __init__(self):
        pass

    def apply(self, result, reference, i_train, i_test):
        result_bool = result.astype(bool)
        reference_bool = reference.astype(bool)
        metrics = []
        for subset, indices in (('train', i_train), ('test', i_test)):
            metrics.append({
                'subset': subset,
                **self.compute_performance_parameters(result_bool[indices], reference_bool[indices]),
            })
        return pd.DataFrame(metrics)

    @staticmethod
    def compute_performance_parameters(result, reference):
        def zero_division(a, b):
            if b != 0:
                return np.round(a / b, 2)
            else:
                return 0.00
        result_not = np.logical_not(result)
        reference_not = np.logical_not(reference)
        tp = np.sum(result[reference])
        fn = np.sum(result_not[reference])
        tn = np.sum(result_not[reference_not])
        fp = np.sum(result[reference_not])
        return {
            'tp': tp,
            'fn': fn,
            'tn': tn,
            'fp': fp,
            'sensitivity': zero_division(tp, tp + fn),

```

```

    'specificity': zero_division(tn, tn + fp),
    'accuracy': zero_division(tp + tn, tp + tn + fn + fp),
    'precision': zero_division(tp, tp + fp)
}

```

The final step before training and evaluating the models is to define the configurations of the different models.

We will train the models with the following configurations:

- Decision tree without features selection
 - Using all the features
 - max_depth: 3
- Decision tree with features selection
 - Using the selected features
 - max_depth: 3
- SVM without features selection
 - Using all the features
 - kernel: rbf
 - gamma: scale
 - regularization: 1
- SVM with features selection
 - Using the selected features
 - kernel: rbf
 - gamma: scale
 - regularization: 1
- Naive Bayes without features selection
 - Using all the features
 - var_smoothing: 1e-09
- Naive Bayes with features selection
 - Using the selected features
 - var_smoothing: 1e-09

```
[11]: exclude_features = []
configs = {
    'decision_tree_all_features': {
        'feature': {
            'list': 'all',
            'exclusion': exclude_features,
            'selection_method': [],
            'selection_numbers': np.nan,
        },
        'model': {
            'name': 'tree',
            'max_depth': 15,
        },
    },
    'decision_tree_selected_features': {

```

```

'feature': {
    'list': features_list,
    'exclusion': exclude_features,
    'selection_method': feature_selection_method,
    'selection_numbers': feature_selection_numbers,
},
'model': {
    'name': 'tree',
    'max_depth': 15,
},
},
'svm_all_features': {
    'feature': {
        'list': 'all',
        'exclusion': exclude_features,
        'selection_method': [],
        'selection_numbers': np.nan,
    },
    'model': {
        'name': 'svm',
        'kernel': 'rbf',
        'gamma': 'scale',
        'regularization': 1,
    },
},
'svm_selected_features': {
    'feature': {
        'list': features_list,
        'exclusion': exclude_features,
        'selection_method': feature_selection_method,
        'selection_numbers': feature_selection_numbers,
    },
    'model': {
        'name': 'svm',
        'kernel': 'rbf',
        'gamma': 'scale',
        'regularization': 1,
    },
},
'bayes_all_features': {
    'feature': {
        'list': 'all',
        'exclusion': exclude_features,
        'selection_method': [],
        'selection_numbers': np.nan,
    },
    'model': {

```

```

        'name': 'bayes',
        'var_smoothing': 1e-09,
    },
},
'bayes_selected_features': {
    'feature': {
        'list': features_list,
        'exclusion': exclude_features,
        'selection_method': feature_selection_method,
        'selection_numbers': feature_selection_numbers,
    },
    'model': {
        'name': 'bayes',
        'var_smoothing': 1e-09,
    },
},
}

```

Now, we are ready to train the models.

```
[12]: models = {}
for name, config in configs.items():
    print(f'* Training {name!r} model')
    model = ModelBuilder(config).apply()
    models[name] = ModelTrainer(config).apply(model, features, targets, i_train)

* Training 'decision_tree_all_features' model
* Training 'decision_tree_selected_features' model
* Training 'svm_all_features' model
* Training 'svm_selected_features' model
* Training 'bayes_all_features' model
* Training 'bayes_selected_features' model
```

Here, we evaluate the trained models on the features.

```
[13]: output = {}
for name, config in configs.items():
    print(f'* Applying {name!r} model')
    output[name] = ModelTester(config).apply(models[name], features)

* Applying 'decision_tree_all_features' model
* Applying 'decision_tree_selected_features' model
* Applying 'svm_all_features' model
* Applying 'svm_selected_features' model
* Applying 'bayes_all_features' model
* Applying 'bayes_selected_features' model
```

Now that all models are trained we can evaluate them on the subsets for training, validation, and testing.

```
[14]: metrics = []
for name, config in configs.items():
    print(f'Evaluating {name} model')
    performance = Evaluator().apply(output[name]['prediction'].values, targets[:, 0], i_train, i_test)
    performance.insert(0, 'model', name)
    metrics.append(performance)
print("\n*** Performance report ***\n")
metrics = pd.concat(metrics, axis=0, ignore_index=True)
metrics = metrics.set_index(['model', 'subset'])
index = metrics.index.get_level_values(0).unique()
columns = pd.MultiIndex.from_product([metrics.columns, metrics.index.
    get_level_values(1).unique()])
metrics = metrics.unstack().reindex(index=index, columns=columns)
IPython.display.display(metrics)
```

Evaluating 'decision_tree_all_features' model
Evaluating 'decision_tree_selected_features' model
Evaluating 'svm_all_features' model
Evaluating 'svm_selected_features' model
Evaluating 'bayes_all_features' model
Evaluating 'bayes_selected_features' model

*** Performance report ***

		tp	fn	tn	fp	
subset	model	train	test	train	test	train
decision_tree_all_features		8047	2593	34	196	6805
decision_tree_selected_features		8055	2578	26	211	6813
svm_all_features		7888	2709	193	80	6282
svm_selected_features		7890	2715	191	74	6290
bayes_all_features		7064	2501	1017	288	4826
bayes_selected_features		7009	2380	1072	409	4792

		sensitivity	specificity	accuracy		
subset	model	train	test	train	test	
decision_tree_all_features		1.00	0.93	0.98	0.96	0.99
decision_tree_selected_features		1.00	0.92	0.98	0.95	0.99
svm_all_features		0.98	0.97	0.91	0.96	0.94
svm_selected_features		0.98	0.97	0.91	0.97	0.95
bayes_all_features		0.87	0.90	0.70	0.84	0.79
bayes_selected_features		0.87	0.85	0.69	0.87	0.79

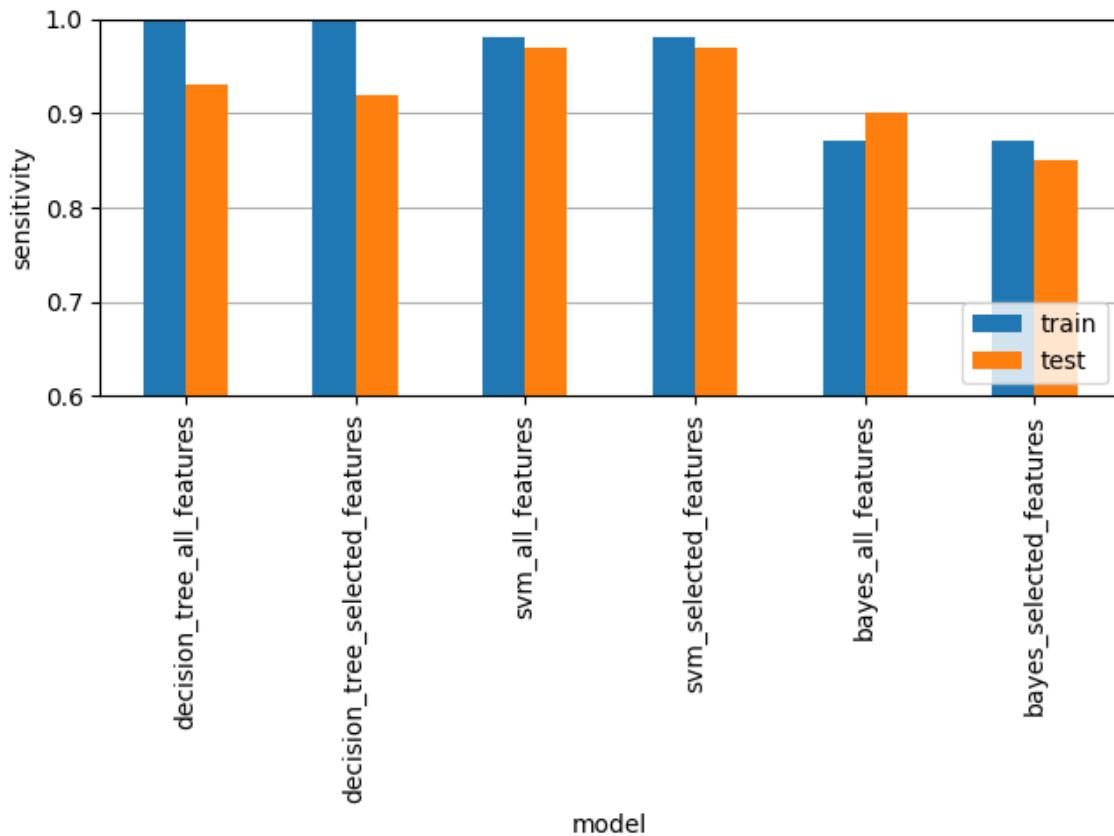
		precision	
subset		test	train

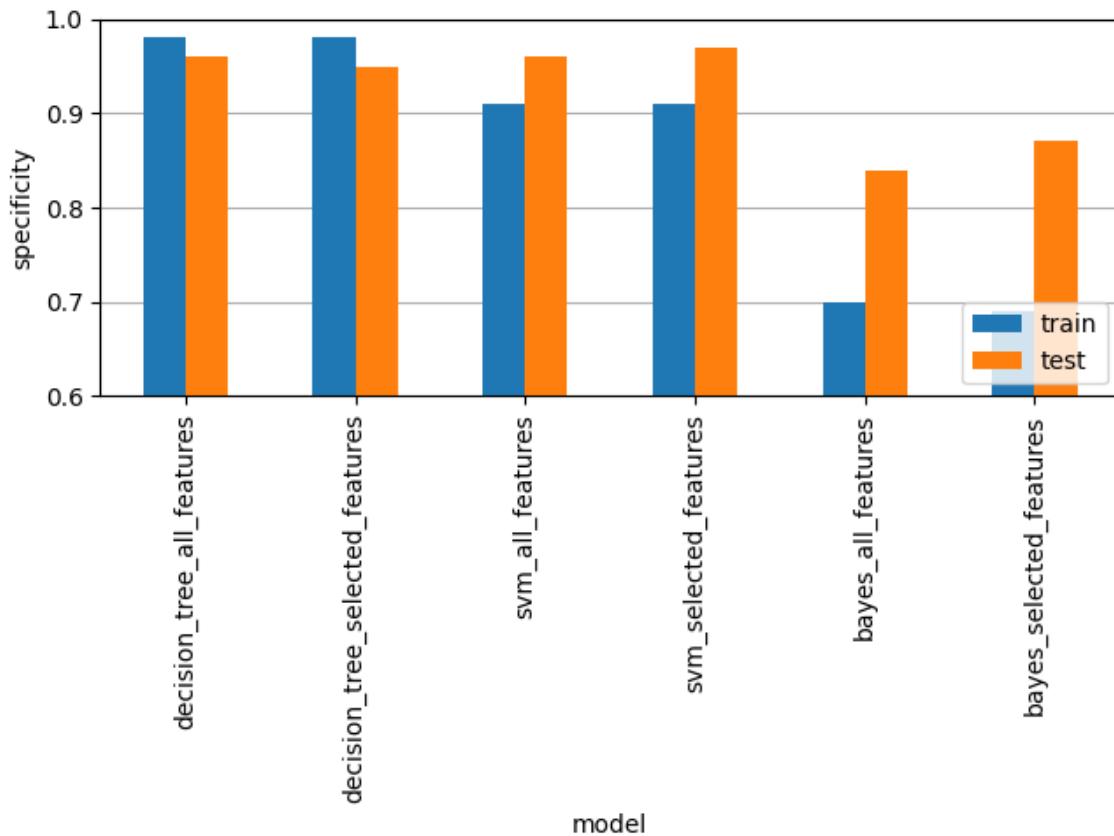
```
model
decision_tree_all_features      0.94      0.99  0.97
decision_tree_selected_features  0.94      0.99  0.96
svm_all_features                0.97      0.93  0.97
svm_selected_features           0.97      0.93  0.97
bayes_all_features              0.87      0.77  0.87
bayes_selected_features         0.86      0.77  0.89
```

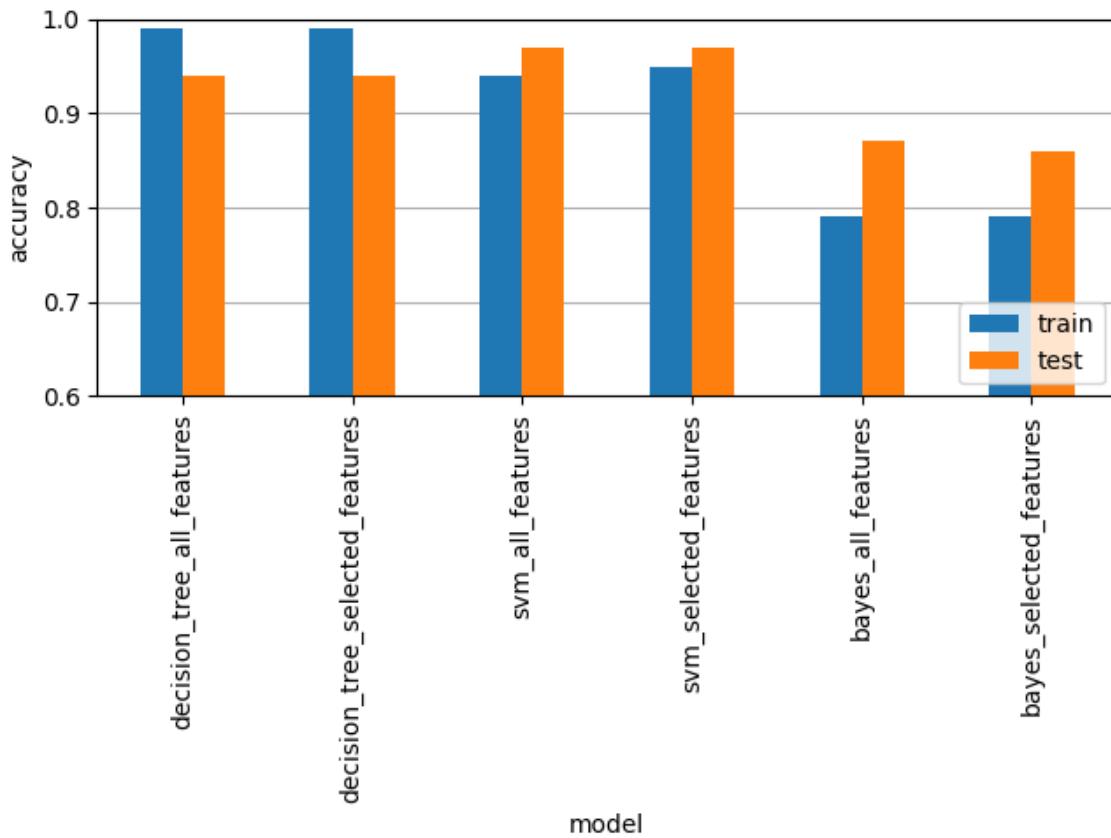
We can also plot the different metrics.

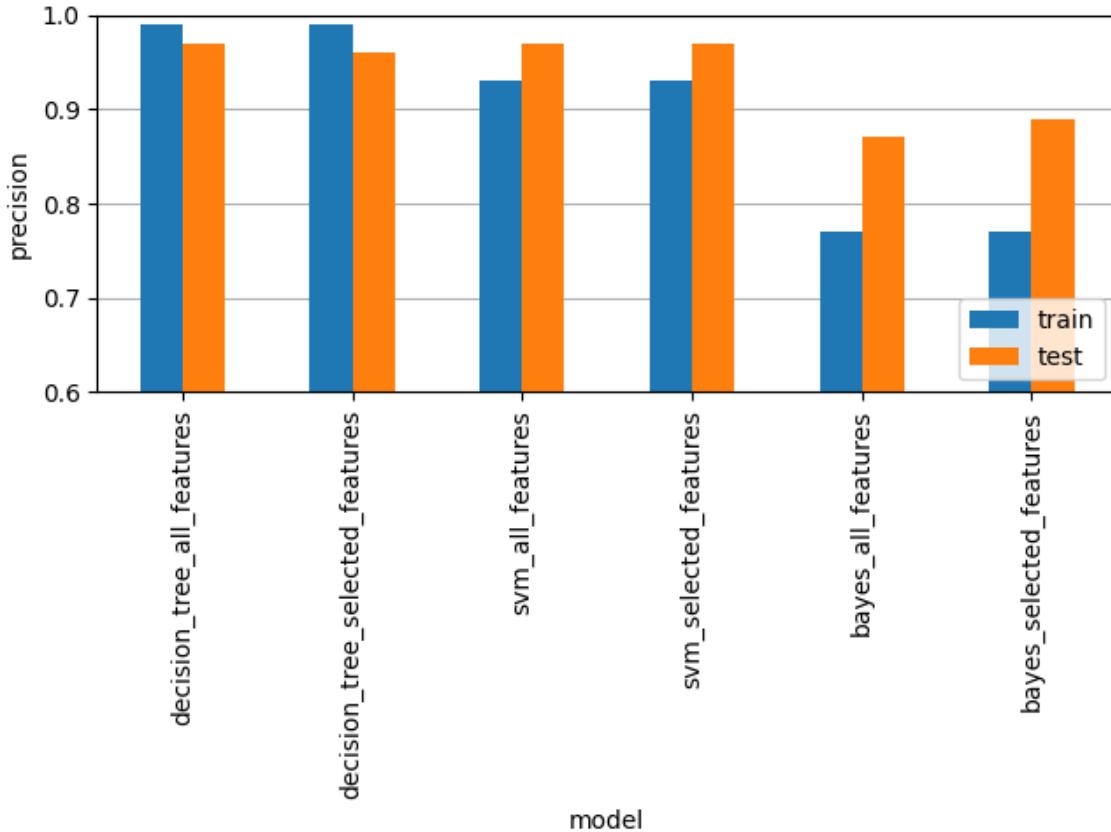
```
[15]: def plot_metrics(data):
    for metric in data.columns.get_level_values(0).unique():
        if metric in ['count', 'tp', 'tn', 'fp', 'fn']:
            continue
        df = data[metric]
        plt.figure(constrained_layout=True)
        plt.gca().set_axisbelow(True)
        df.plot(kind='bar', ylabel=metric, ax=plt.gca())
        plt.grid(axis='y')
        plt.ylim(0.6, 1.0)
        plt.legend(loc='lower right')
        plt.gca().xaxis.set_tick_params(rotation=90)

plot_metrics(metrics)
```









2 Exercise Questions

2.1 Question 6

If you want to select a set of features manually, which features would you choose and why?

Answer

Using the criteria written down in Question 1, one would first look for features that have their class density distribution well separated from one another (in simpler terms, the graphs on the diagonal where the orange plot overlaps the blue one as less as possible), and then choose the features that have low correlations with one another.

Using these facts, a wise choice of features would be mean, std and p2p.

2.2 Question 7

Do the automatically selected features match your manually selected features? Explain the reasons for any similarities and/or differences.

Answer

The results are again quite similar between the manually and automatically selected features.

The automatic selection used “LASSO” feature selection, which works by introducing a penalty term into the regression model that shrinks the coefficients of less important features towards zero, effectively removing them from the model. The similarities are that in both cases the correlation of the features with the target classes, as well as the correlations between them. However, Lasso takes it a bit further as it considers the correlation between multiple features. So it can find potential relations that would not appear on the two-dimensional pair plot representation. On the other hand, manually, you might rule out certain features that have no physiological meaning, which Lasso cannot do.

2.3 Question 8

Do you see any signs of overfitting and/or underfitting of the models? Why?

Answer

Over-fitting is highlighted by the test error being much higher than the training error. Looking at Figure 5, the decision tree classifier is such a case of over-fitting, regardless of the feature selection, indicating that the model has been trained for too long, becoming a bit too specific to the training data and performing worse on unseen data.

Under-fitting occurs when both test and train errors are high. Looking at Figure 5, such a case is shown for the Bayes model, again regardless of the feature selection, this indicates that the model is too simplistic to capture the underlying pattern in the data. A possible reason for this bad performance might be the fact that the Bayes model assumes independence between the features.

2.4 Question 9

Considering all conditions, which model will you finally choose to detect atrial fibrillation? Why?

Answer

Considering all conditions, it is clear that the Naive Bayes model is not the right choice, as it performs significantly worse than the other two models.

In this case, the best model to use would be the SVM with only selected features. Even though both models have similar performances (same accuracy, specificity, ...), the one with less features is also faster in terms of computation time. Moreover, the sensitivity metric is the most important one to consider as it has much worse consequences to miss positive cases than to falsely identify someone as positive in this case. Furthermore, this model does not show any sign of over-/under-fitting. Therefore SVM with only selected features is the best model to choose.

hr_estimation

December 5, 2024

1 Heart Rate Estimation

The goal of this exercise is to estimate the heart rate from PPG and acceleration signals using regression methods. We use data from the PPG-DaLiA dataset (<https://archive.ics.uci.edu/ml/datasets/PPG-DaLiA>). It includes PPG and acceleration signals as well as the reference heart rate computed from an ECG signal. These signals were collected during various activity but we focus on two of them: sitting and walking.

First, we import all the packages we will need, define some global variables, and seed the random number generators.

```
[18]: %matplotlib widget

import copy
import itertools
import operator
import pathlib
import warnings
import IPython.display
import joblib
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import f_regression
from sklearn.feature_selection import SelectFromModel
from sklearn.linear_model import Lasso
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

DATA_FILE = pathlib.Path('../data/ppg_dalia.pkl')
LOG_DIRECTORY = pathlib.Path('../logs/hr_estimation')
```

Then, we load the PPG and acceleration signals as well as the reference heart rate. The signals are

already pre-processed with the following steps:

- Band-pass filtering between 0.4 and 4.0 Hz (24 - 240 bpm).
- Resampling to 25 Hz.

We also define the window length and shift length used to compute the reference heart rate.

```
[19]: FS = 25.0 # Sampling frequency of the PPG and acceleration signals in Hertz.  
WINDOW_LENGTH = 8.0 # Window duration in seconds used to compute the reference  
# heart rate.  
SHIFT_LENGTH = 2.0 # Shift between successive windows in seconds.  
  
WINDOW_SIZE = round(FS * WINDOW_LENGTH)  
SHIFT_SIZE = round(FS * SHIFT_LENGTH)  
  
records = joblib.load(DATA_FILE)  
subjects = set(record['subject'] for record in records)  
  
print(f'Window length: {WINDOW_LENGTH} s (n = {WINDOW_SIZE})')  
print(f'Shift length: {SHIFT_LENGTH} s (n = {SHIFT_SIZE})')  
print(f'Number of records: {len(records)}')  
print(f'Number of subjects: {len(subjects)}')
```

Window length: 8.0 s (n = 200)

Shift length: 2.0 s (n = 50)

Number of records: 29

Number of subjects: 15

Here are two examples of PPG and acceleration signals. One recorded when the subject is sitting and one recorded when the subject is walking.

```
[20]: def plot_signals(record):  
    signals = record['signals']  
    hr = record['hr']  
  
    fig, axes = plt.subplots(3, 1, sharex='all', constrained_layout=True)  
    plt.suptitle(f'{record["subject"]} ({record["activity"]})')  
  
    plt.sca(axes.flat[0])  
    plt.plot(signals['time'].to_numpy(),  
             signals[['acc_x', 'acc_y', 'acc_z']].to_numpy(),  
             linewidth=1)  
    plt.grid()  
    plt.ylabel('Acceleration')  
  
    plt.sca(axes.flat[1])  
    plt.plot(signals['time'].to_numpy(), signals['ppg'].to_numpy(),  
             linewidth=1)  
    plt.grid()
```

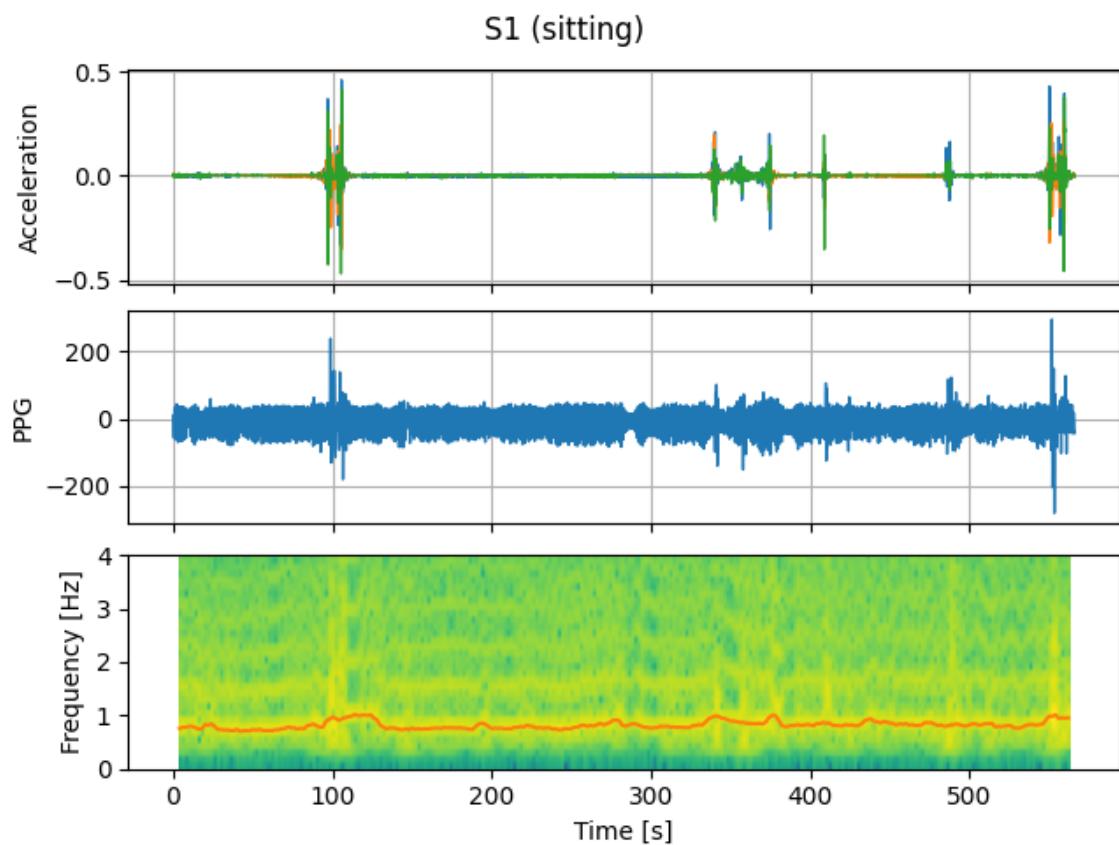
```

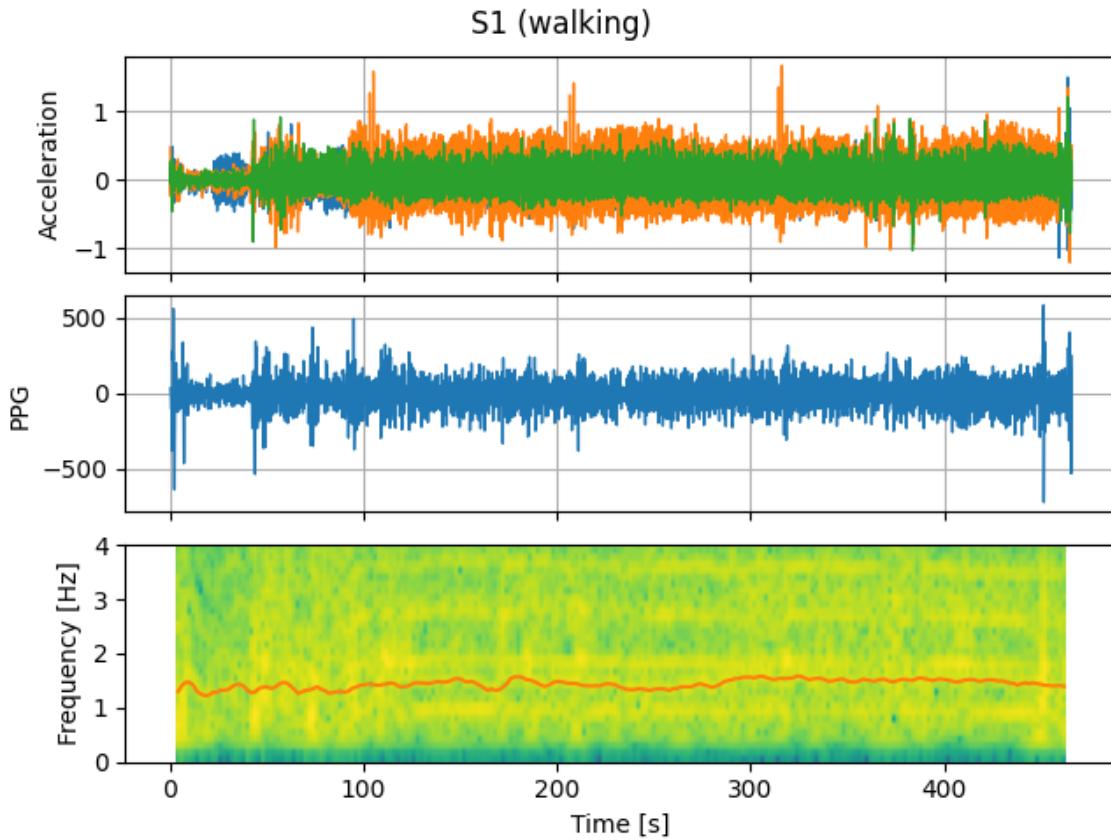
plt.ylabel('PPG')

plt.sca(axes.flat[2])
plt.specgram(signals['ppg'].to_numpy(), Fs=FS, NFFT=WINDOW_SIZE,
             nooverlap=WINDOW_SIZE - SHIFT_SIZE)
plt.plot(hr['time'].to_numpy(), hr['hr'].to_numpy() / 60.0,
         color='tab:orange')
plt.ylim(0.0, 4.0)
plt.xlabel('Time [s]')
plt.ylabel('Frequency [Hz]')

plot_signals(records[0])
plot_signals(records[1])

```





By zooming on the PPG signal, it is clear that walking cause a degradation in signal quality.

We will try to estimate the heart rate on sliding windows of the PPG and acceleration signals. To make things easier, we use the same window length and shift between windows as the reference heart rate.

So the next step is to extract sliding windows from all the records. We also extract the corresponding subject identifier for splitting the windows into subsets for training, validation, and testing.

In addition, we also prepare windows that include only the PPG signal (first channel).

```
[21]: def extract_windows(record):
    x = record['signals'][['ppg', 'acc_x', 'acc_y', 'acc_z']].to_numpy()
    n = x.shape[0]

    windows = []
    for i, start in enumerate(range(0, n - WINDOW_SIZE + 1, SHIFT_SIZE)):
        end = start + WINDOW_SIZE
        windows.append(x[start:end].T)
    windows = np.stack(windows)
    targets = record['hr']['hr'].to_numpy()
```

```

    return windows, targets

def extract_all_windows(records):
    windows = []
    targets = []
    subjects = []
    activities = []
    for record in records:
        x, y = extract_windows(record)
        windows.append(x)
        targets.append(y)
        subjects.extend(itertools.repeat(record['subject'], x.shape[0]))
        activities.extend(itertools.repeat(record['activity'], x.shape[0]))

    windows = np.concatenate(windows, axis=0)
    targets = np.concatenate(targets)[:, None]
    subjects = np.array(subjects)
    activities = np.array(activities)

    return windows, targets, subjects, activities

ppg_acc_windows, targets, subjects, activities = extract_all_windows(records)
ppg_windows = ppg_acc_windows[:, 0, :]

print(f'Shape of PPG and acceleration windows: {ppg_acc_windows.shape}')
print(f'Shape of PPG windows: {ppg_windows.shape}')

```

Shape of PPG and acceleration windows: (7420, 4, 200)
Shape of PPG windows: (7420, 200)

We have 7420 windows with 1 or 4 channels and that each window includes 200 samples (8 seconds at 25 Hz).

Next, we split the windows for training, validation, and testing by subjects. The test set includes 9 subjects, the validation set 3 subjects, and the test set 3 subjects.

```
[22]: def split_subjects(subjects):
    val_subjects = ('S10', 'S11', 'S12')
    test_subjects = ('S13', 'S14', 'S15')

    i_val = np.flatnonzero(np.isin(subjects, val_subjects))
    i_test = np.flatnonzero(np.isin(subjects, test_subjects))
    i_train = np.setdiff1d(np.arange(subjects.size), np.union1d(i_val, i_test))

    assert not (set(subjects[i_train]) & set(subjects[i_val]))
    assert not (set(subjects[i_train]) & set(subjects[i_test]))
```

```

    assert not (set(subjects[i_val]) & set(subjects[i_test]))
    assert (set(subjects[i_train]) | set(subjects[i_val]) | 
    ~set(subjects[i_test])) == set(subjects)

    return i_train, i_val, i_test

i_train, i_val, i_test = split_subjects(subjects)

print(f'Subject used for training : {pd.unique(subjects[i_train])}')
print(f'Subject used for validation : {pd.unique(subjects[i_val])}')
print(f'Subject used for testing : {pd.unique(subjects[i_test])}')

```

Subject used for training : ['S1' 'S2' 'S3' 'S4' 'S5' 'S6' 'S7' 'S8' 'S9']
 Subject used for validation : ['S10' 'S11' 'S12']
 Subject used for testing : ['S13' 'S14' 'S15']

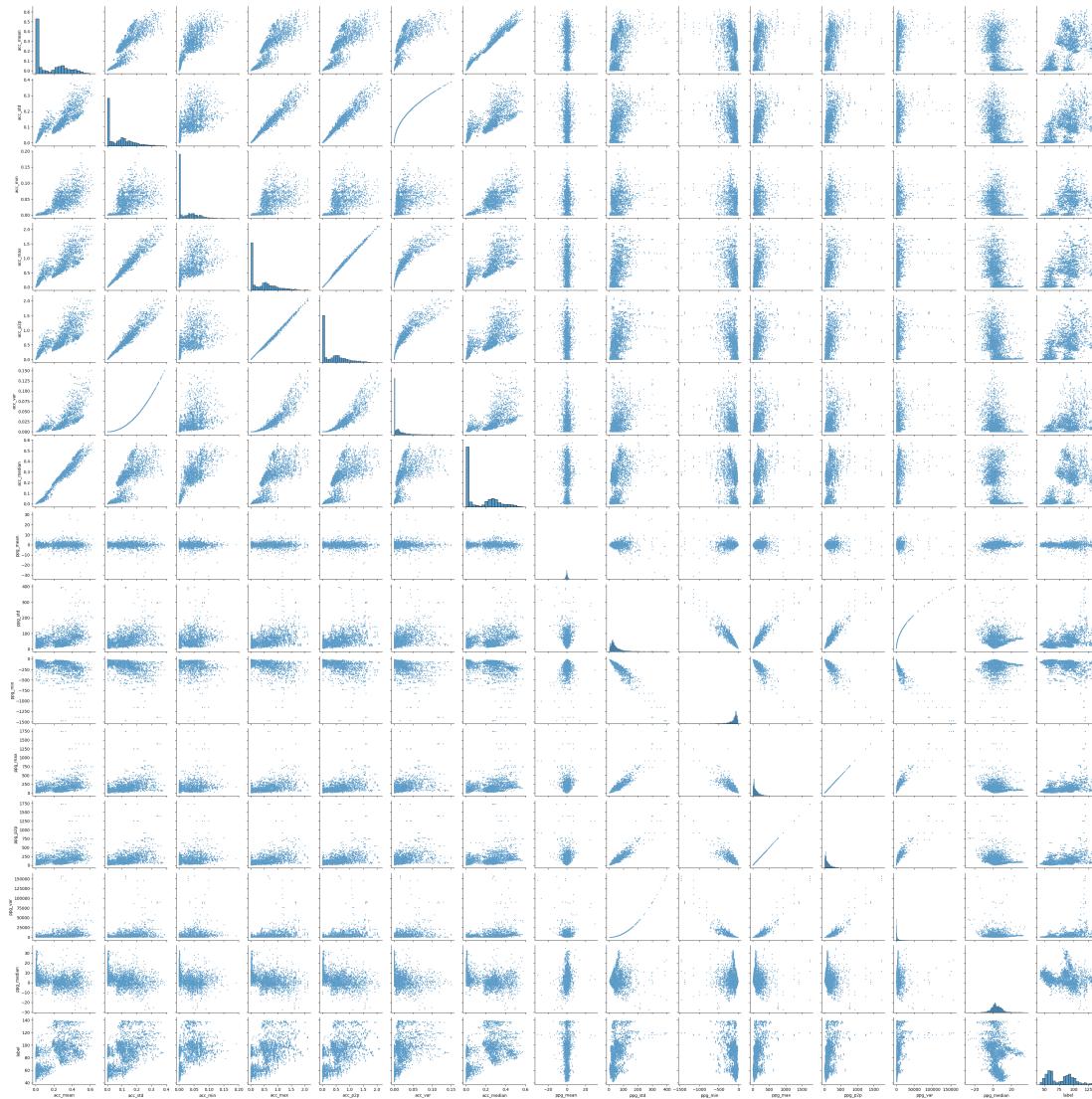
Now we extract features from the windows.

```
[23]: acc_norm = np.zeros([ppg_acc_windows.shape[0], ppg_acc_windows.shape[2]])
for instance in range(ppg_acc_windows.shape[0]):
    ppg_acc = ppg_acc_windows[instance, :, :]
    acc_norm[instance, :] = np.sqrt(np.sum(np.power(ppg_acc, 2), axis=0))

features = pd.DataFrame({
    'acc_mean': np.mean(acc_norm, axis=1),
    'acc_std': np.std(acc_norm, axis=1),
    'acc_min': np.min(acc_norm, axis=1),
    'acc_max': np.max(acc_norm, axis=1),
    'acc_p2p': np.max(acc_norm, axis=1) - np.min(acc_norm, axis=1),
    'acc_var': np.power(np.std(acc_norm, axis=1), 2),
    'acc_median': np.median(acc_norm, axis=1),
    'ppg_mean': np.mean(ppg_windows, axis=1),
    'ppg_std': np.std(ppg_windows, axis=1),
    'ppg_min': np.min(ppg_windows, axis=1),
    'ppg_max': np.max(ppg_windows, axis=1),
    'ppg_p2p': np.max(ppg_windows, axis=1) - np.min(ppg_windows, axis=1),
    'ppg_var': np.power(np.std(ppg_windows, axis=1), 2),
    'ppg_median': np.median(ppg_windows, axis=1),
})

def plot_features(f, y):
    data = f.copy()
    data['label'] = y.ravel()
    sns.pairplot(data, plot_kws={'s': 4})
```

```
plot_features(features.iloc[i_train], targets[i_train])
```



It is also possible to select the most relevant features using various methods. Here, we define implement three fature selection techniques: lasso, univariate, hybrid.

```
[24]: class FeatureSelector:

    def __init__(self, method, numbers):
        self.method = method.lower()
        self.numbers = numbers

    def apply(self, features, targets):
```

```

        features_names = [column for column in features.columns
                           if column not in ['reference']]
        features_selection = features.copy()
        features_selection.insert(0, 'reference', targets)
        features_selection = features_selection.dropna(axis=0, how='any', ↴
inplace=False)
        ranks = self.rank_features(features_selection[features_names],
                                    features_selection['reference'],
                                    self.method)
    del features_selection
    return self.select_features(ranks, self.numbers)

    @staticmethod
    def select_features(ranks, feature_num):
        ranks.sort_values(by='ranks', axis=0, ascending=False, inplace=True,
                          kind='quicksort', ignore_index=True)
        return ranks['feature_names'].iloc[: feature_num].tolist()

    @staticmethod
    def rank_features(features, reference, method):
        def univariate_selection(data, ref):
            selector = SelectKBest(f_regression, k="all")
            scores = selector.fit(data, ref).scores_
            return scores / np.nansum(scores)

        def lasso_selection(data, ref):
            alphas = np.arange(0.01, 0.3, 0.01)
            coefficients = np.empty([len(alphas), data.shape[1]])
            for row, alpha in enumerate(alphas):
                selector = SelectFromModel(Lasso(alpha=alpha), prefit=False)
                coefficients[row, :] = selector.fit(data, ref).estimator_.coef_
            coefficients = np.abs(coefficients)
            real_ranks = np.nansum(coefficients, axis=0)
            return real_ranks / np.nansum(real_ranks)

        if method == 'lasso':
            ranks = lasso_selection(features, reference)
        elif method == 'univariate':
            ranks = univariate_selection(features, reference)
        elif method == 'hybrid':
            rank_lasso = lasso_selection(features, reference)
            rank_univariate = univariate_selection(features, reference)
            rank_combined = rank_lasso + rank_univariate
            ranks = rank_combined / np.nansum(rank_combined)
        else:
            raise TypeError("Feature selection method is not supported")
        return pd.DataFrame({

```

```

        'feature_names': features.columns,
        'ranks': ranks,
    })

feature_selection_method = 'univariate'
feature_selection_numbers = 4
features_list = FeatureSelector(feature_selection_method, ↴
    ↪feature_selection_numbers).apply(features.iloc[i_val], targets[i_val])
print(f"Selected features:{features_list}")

```

Selected features: ['acc_mean', 'acc_median', 'acc_std', 'acc_min']

Now we define the regression models: linear regression, support vector regression.

```
[25]: class ModelBuilder:

    def __init__(self, config):
        self.config = config['model']

    def apply(self):
        return eval(f"self._build_{self.config['name']}()")

    def _build_lregression(self):
        return make_pipeline(
            StandardScaler(),
            LinearRegression())

    def _build_svr(self, kernel='rbf', gamma='scale', regularization=1):
        if 'kernel' in self.config.keys():
            kernel = self.config['kernel']
        if 'gamma' in self.config.keys():
            gamma = self.config['gamma']
        if 'regularization' in self.config.keys():
            regularization = self.config['regularization']
        return make_pipeline(
            StandardScaler(),
            SVR(kernel=kernel, gamma=gamma, C=regularization))
```

Now, we define a class for the training of the models.

```
[26]: class ModelTrainer:

    def __init__(self, config):
        self.config = config['feature']

    def apply(self, model, features, reference, i_train):
        features_list = self._get_features_list(list(features.columns))
        features_train = features.copy()
```

```

        features_train.insert(0, 'reference', reference)
        features_train = features_train.iloc[i_train].copy()

        features_train = features_train.dropna(axis=0, how='any', inplace=False,
                                              subset=features_list + [
→['reference']])
        return model.fit(
            features_train[features_list].values, features_train['reference'].
→values)

    def _get_features_list(self, current_features):
        if 'all' in self.config['list']:
            features_list = [feature for feature in current_features
                            if feature not in self.config['exclusion']] +
→['reference']]
        else:
            features_list = [feature for feature in
                            self.config['list']
                            if feature in current_features and feature not
                            in self.config['exclusion']] + ['reference']
    return features_list

```

We also define a class to apply the trained models on the test data.

```
[27]: class ModelTester:

    def __init__(self, config):
        self.config = config['feature']

    def apply(self, model, features):
        features_list = self._get_features_list(list(features.columns))
        inx = np.logical_not(
            np.sum(np.isnan(features[features_list]), 1).astype(bool))
        detections = np.zeros_like(inx, dtype=float)
        detections[:] = np.nan
        detections[inx] = model.predict(features[features_list].values[inx])
        return pd.DataFrame({
            'prediction': detections,
        })

    def _get_features_list(self, current_features):
        if 'all' in self.config['list']:
            features_list = [feature for feature in current_features
                            if feature not in self.config['exclusion']] +
→['reference']]
        else:
            features_list = [feature for feature in

```

```

        self.config['list']
        if feature in current_features and feature not
        in self.config['exclusion'] + ['reference']]
    return features_list

```

Inorder to evaluate the results of the models, we define an Evaluator class as follows:

```
[28]: class Evaluator:

    def __init__(self):
        pass

    def apply(self, result, reference, i_train, i_test):
        metrics = []
        for subset, indices in (('train', i_train), ('test', i_test)):
            metrics.append({
                'subset': subset,
                **self.compute_performance_parameters(result[indices], ↴
reference[indices]),
            })
        return pd.DataFrame(metrics)

    @staticmethod
    def compute_performance_parameters(result, reference):
        error = np.zeros_like(reference)
        error[:] = np.nan
        inx = reference != 0
        error[inx] = 100 * ((reference[inx] - result[inx]) / reference[inx])
        error = error[error != np.nan]
        return {
            'mean': np.mean(error),
            'std': np.std(error),
            'rmse': np.sqrt(np.mean(np.power(error, 2))),
        }
```

The final step before training and evaluating the models is to define the configurations of the different models.

We will train the models with the following configurations:

- Linear without features selection
 - Using all the features
- Linear regression with features selection
 - Using the selected features
- SVR with features selection
 - Using the selected features
 - kernel: rbf
 - gamma:scale
 - regularization: 1

```
[29]: exclude_features = []
configs = {
    'linear_regression_all_features': {
        'feature': {
            'list': 'all',
            'exclusion': exclude_features,
            'selection_method': [],
            'selection_numbers': np.nan,
        },
        'model': {
            'name': 'lregression',
        },
    },
    'linear_regression_selected_features': {
        'feature': {
            'list': features_list,
            'exclusion': exclude_features,
            'selection_method': feature_selection_method,
            'selection_numbers': feature_selection_numbers,
        },
        'model': {
            'name': 'lregression',
        },
    },
    'svr_all_features': {
        'feature': {
            'list': 'all',
            'exclusion': exclude_features,
            'selection_method': [],
            'selection_numbers': np.nan,
        },
        'model': {
            'name': 'svr',
            'kernel': 'rbf',
            'gamma': 'scale',
            'regularization': 1,
        },
    },
    'svr_selected_features': {
        'feature': {
            'list': features_list,
            'exclusion': exclude_features,
            'selection_method': feature_selection_method,
            'selection_numbers': feature_selection_numbers,
        },
        'model': {
            'name': 'svr',
```

```

        'kernel': 'rbf',
        'gamma': 'scale',
        'regularization': 1,
    },
},
}

```

Now, we are ready to train the models.

```
[30]: models = {}
for name, config in configs.items():
    print(f'* Training {name!r} model')
    model = ModelBuilder(config).apply()
    models[name] = ModelTrainer(config).apply(model, features, targets, i_train)

* Training 'linear_regression_all_features' model
* Training 'linear_regression_selected_features' model
* Training 'svr_all_features' model
* Training 'svr_selected_features' model
```

Here, we evaluate the trained models on the features.

```
[31]: output = {}
for name, config in configs.items():
    print(f'* Applying {name!r} model')
    output[name] = ModelTester(config).apply(models[name], features)

* Applying 'linear_regression_all_features' model
* Applying 'linear_regression_selected_features' model
* Applying 'svr_all_features' model
* Applying 'svr_selected_features' model
```

Now that all models are trained we can evaluate them on the subsets for training, validation, and testing.

```
[32]: metrics = []
for name, config in configs.items():
    print(f'Evaluating {name!r} model')
    performance = Evaluator().apply(output[name]['prediction'].values, targets[:,
        0], i_train, i_test)
    performance.insert(0, 'model', name)
    metrics.append(performance)
print("\n*** Performance report ***\n")
metrics = pd.concat(metrics, axis=0, ignore_index=True)
metrics = metrics.set_index(['model', 'subset'])
index = metrics.index.get_level_values(0).unique()
columns = pd.MultiIndex.from_product([metrics.columns, metrics.index.
    get_level_values(1).unique()])
metrics = metrics.unstack().reindex(index=index, columns=columns)
```

```

In [32]: IPython.display.display(metrics)

Evaluating 'linear_regression_all_features' model
Evaluating 'linear_regression_selected_features' model
Evaluating 'svr_all_features' model
Evaluating 'svr_selected_features' model

```

*** Performance report ***

	mean		std	\
subset	train	test	train	
model				
linear_regression_all_features	-3.924053	-12.435107	19.565851	
linear_regression_selected_features	-4.549709	-8.650772	20.517729	
svr_all_features	0.418621	-7.181589	16.625634	
svr_selected_features	1.217230	-2.908962	17.959907	
		rmse		
subset	test	train	test	
model				
linear_regression_all_features	17.199001	19.955468	21.223514	
linear_regression_selected_features	16.334608	21.016114	18.483919	
svr_all_features	15.572246	16.630904	17.148471	
svr_selected_features	10.697804	18.001109	11.086256	

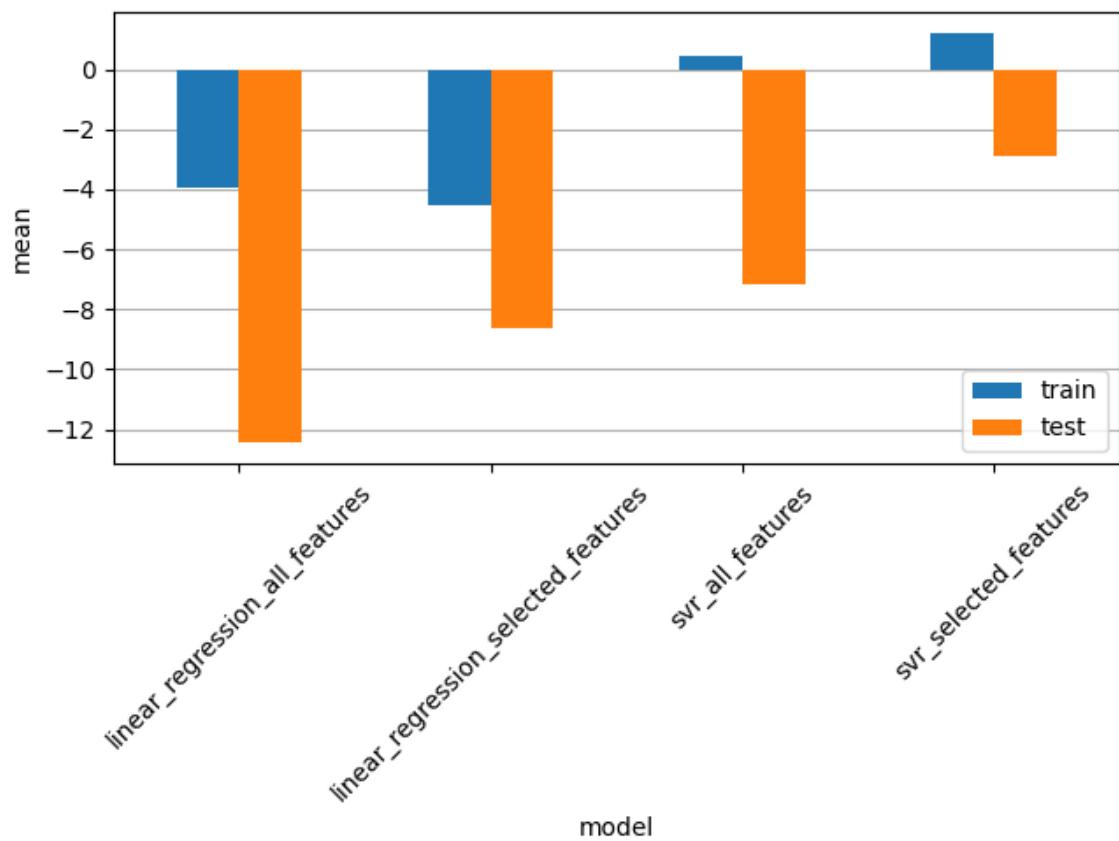
We can also plot the different metrics.

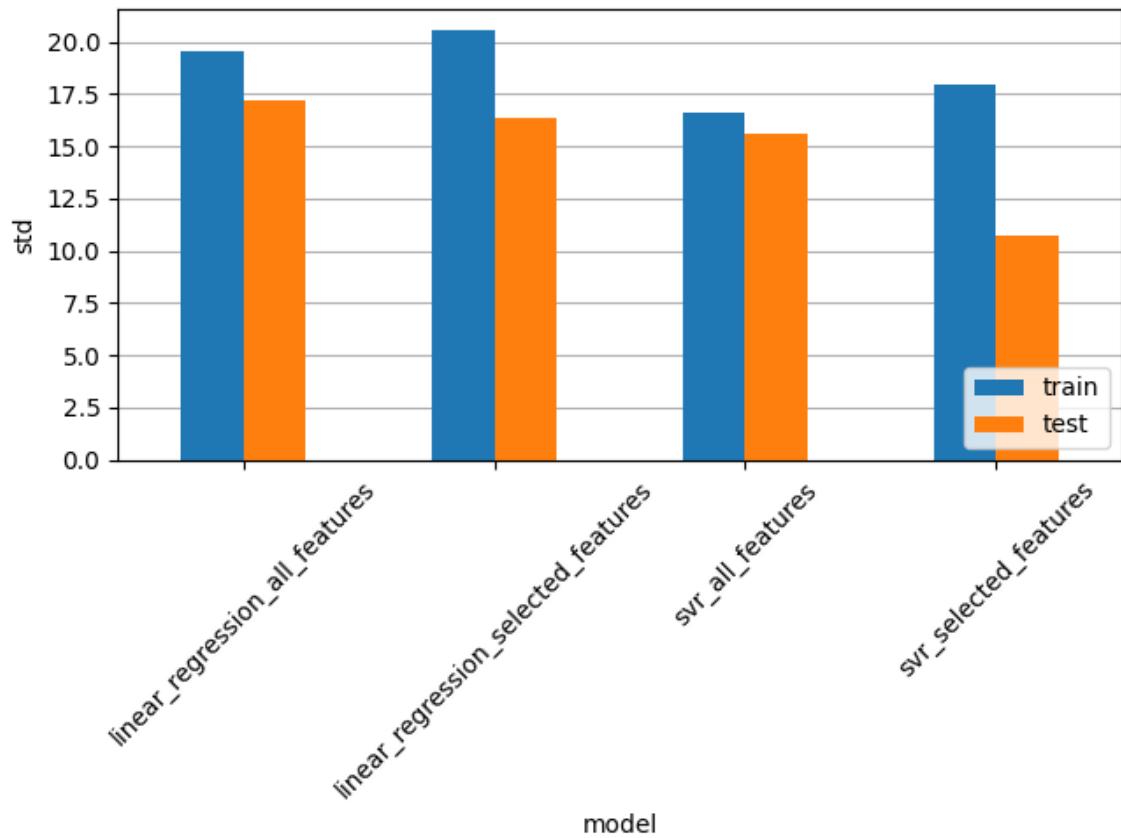
```

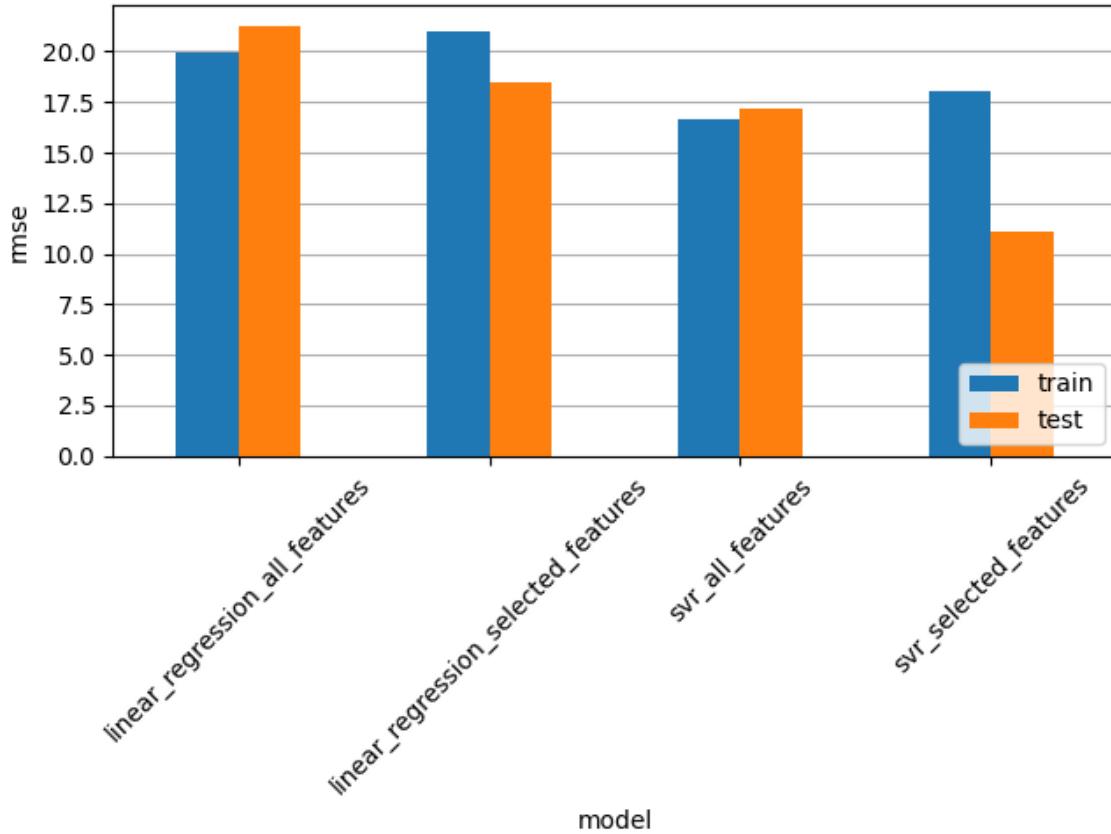
[33]: def plot_metrics(data):
    for metric in data.columns.get_level_values(0).unique():
        if metric == 'count':
            continue
        df = data[metric]
        plt.figure(constrained_layout=True)
        plt.gca().set_axisbelow(True)
        df.plot(kind='bar', ylabel=metric, ax=plt.gca())
        plt.grid(axis='y')
        plt.legend(loc='lower right')
        plt.gca().xaxis.set_tick_params(rotation=45)

plot_metrics(metrics)

```







2 Exercise Questions

2.1 Question 1

If you want to select a set of features manually, which features would you choose and why?

Answer

Generally, one would choose the features that are the most important to explain all the different data available. For example, if one notices a strong correlation between the change of the value of one feature and the output, it is optimal to select such a feature.

Another important fact to take into account is the independency of the features : if one feature is strongly correlated with another, there is no use in keeping both features in the model, since one of them is enough to illustrate the behaviour of the other.

Taking these facts into account, one can see that the labels are not very much correlated with the PPG information (the last column/last row plots in Figure 3 all have a low correlation). On the other hand, the accelerometer data are much more correlated; taking into account the fact that the correlation between features should be kept at its lowest, a good feature choice - if we had to take three of them - would be acc_var, acc_mean and acc_min for example.

2.2 Question 2

Do the automatically selected features match your manually selected features? Explain the reasons for any similarities and/or differences.

Answer

The automatically selected features are similar to our choice : acc_var is directly proportionnal to the square of acc_std, therefore both are related.

The reason why the features match is because the automatically selected features are chosen using [SelectKBest](#), which uses the ANOVA F-value to extract the score of each feature, and this is based on the criteria we wrote down in Question 1. Our manual choice was probably less precise since it was only based on the graphs, but it does not change much in the end, since the features are meaningful and easily understandable by humans.

2.3 Question 3

Do you think that the feature selection was useful in this exercise? Why?

Answer

Looking at Figures 4, 5 and 6, the mean error, standard deviation error RMSE of the test data all decrease for the test data when using only selected features, showing an overall better performance of the model, while the model and computational complexity were successfully reduced.

2.4 Question 4

How do you interpret mean, std, and rmse errors of the models?

Answer

The mean error can be interpreted as the following: the average “signed-distance” from which our predictions are to their real values.

The standard deviation error can be considered as the variability of the prediction to the real values. Predicted values are different to the real values, however, if we take a 2D plane, the predictions can tend to always be different by 2 on the y-axis and -1 on the x-axis. The lower the STD, the more similar the errors between predictions are

The Root Mean Squared Error is the average “squared-distance” from which our predictions are to their real values, it indicates the overall performance of the model.

2.5 Question 5

Considering all conditions, which model will you finally choose to estimate heart rate? Why?

Answer

To choose the best model, we would ideally take the model presenting the smallest metrics for absolute, std and rmse. In all three metrics, the linear regression models express higher values. Which is why we choose one of the SVR models. SVR using only the selected features, although the train error metrics are slightly higher than with all features, it seems to give the best test results when considering the test error metrics, and it also reduces the computational costs compared to the SVR using all the features.