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.

Group Members: Mamoun Alaoui Slimani, Léa Slive, Yasaman Noorikhah, Antonio Del Priore Antunes
Date of Submission: Thursday, October 2, 2025

Please find the answer of each question written as a markdown or code cell after the question.

First we need to install the required python packages as follows:

In [1]:
```
# pip install -r "requirements.txt"
```

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

In [2]:
```
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
%matplotlib widget

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.

In [3]:
```
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: {np.str_('normal_sinus_rhythm'), np.str_('atrial_fibrillation')}
```

Here are a few examples of windows of interbeat intervals.

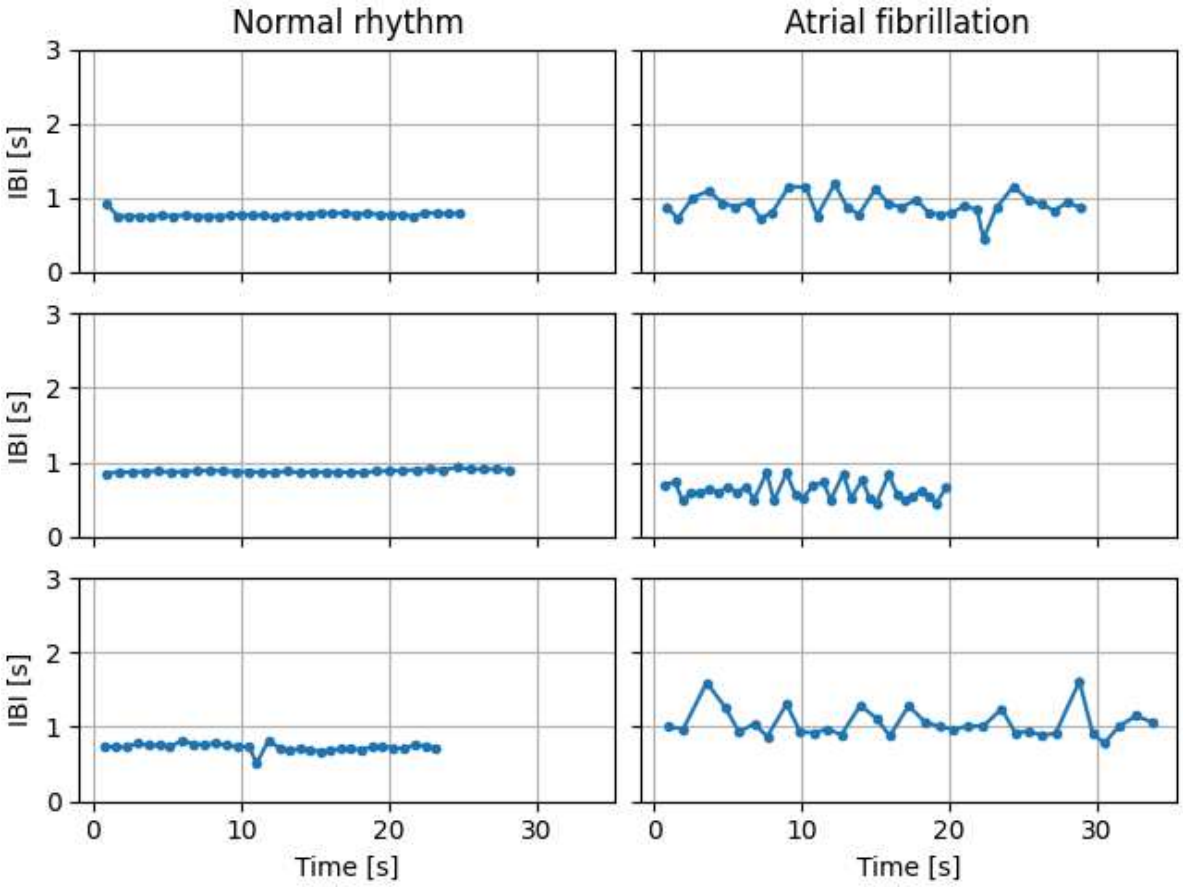
In [4]:
```
def plot_interval_examples(intervals, targets, n_examples=3):
    normal_indices = np.random.choice(np.flatnonzero(targets == 0.0), n_examples, r
    af_indices = np.random.choice(np.flatnonzero(targets == 1.0), n_examples, repla

    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=Fal
    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)
```

Figure



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

In [5]:
```
def split_data(identifiers, intervals, targets):
    splitter = sklearn.model_selection.StratifiedGroupKFold(n_splits=5)
    indices = list(map(operator.itemgetter(1), splitter.split(intervals, targets, i
    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_

    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'), (targe
```

	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.

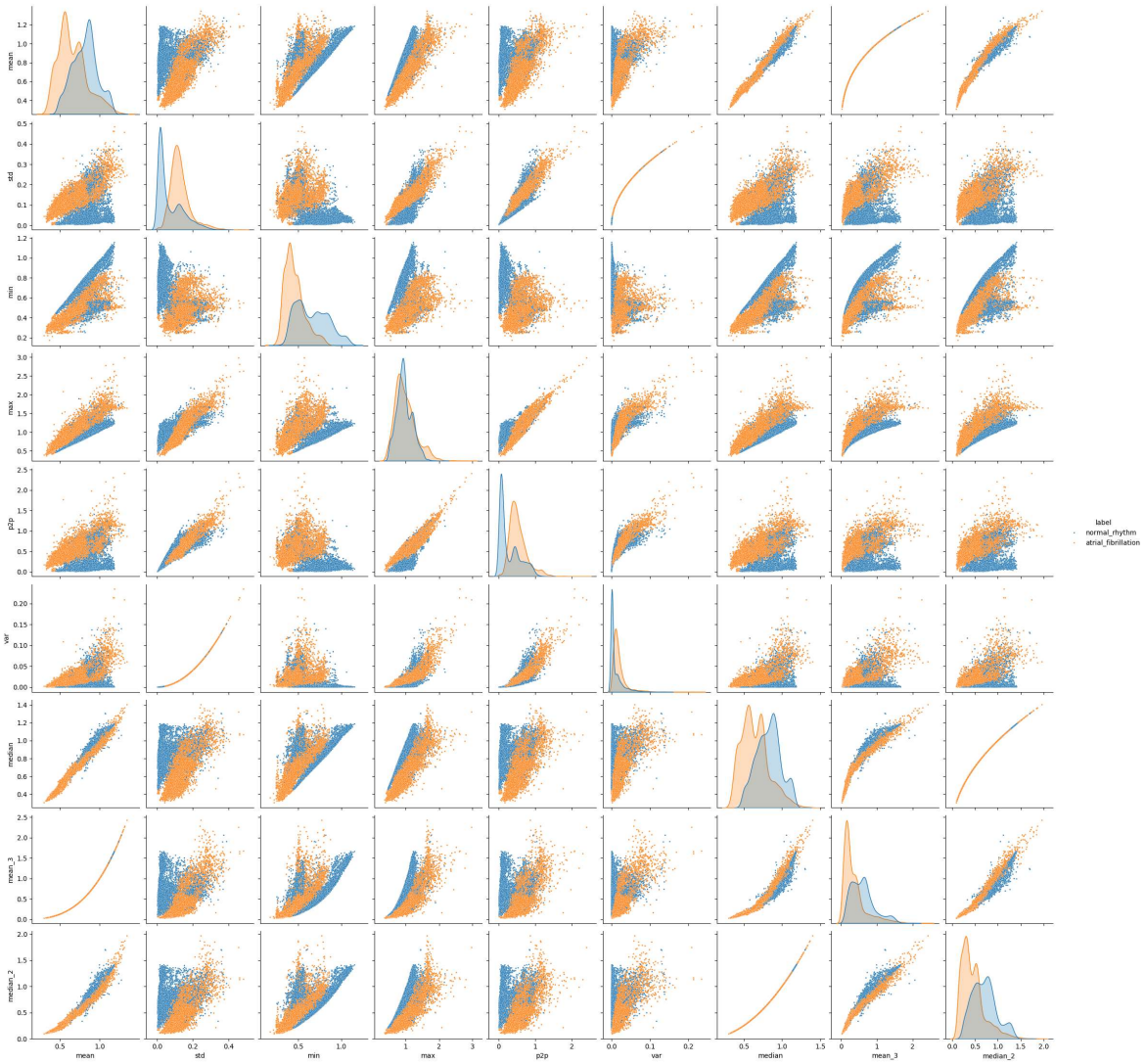
In [6]:
```
f_mean = np.mean(intervals, axis=1)
f_std = np.std(intervals, axis=1)
f_min = np.min(intervals, axis=1)
f_max = np.max(intervals, axis=1)
f_median = np.median(intervals, axis=1)
features = pd.DataFrame({
    'mean': f_mean,
    'std': f_std,
    'min': f_min,
    'max': f_max,
    'p2p': 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: 'atrial_fibrillat
    sns.pairplot(data, hue='label', plot_kws={'s': 4})


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

Figure



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

In [7]:
```
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 ['
        features_selection = features.copy()
        features_selection.insert(0, 'reference', targets)
        features_selection = features_selection.dropna(axis=0, how='any', inplace=F
        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 = 'lasso'
feature_selection_numbers = 5
features_list = FeatureSelector(feature_selection_method, feature_selection_numbers
print(f"Selected features:{features_list}")
```

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

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.

In [8]:
```
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.

In [9]:

```
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'].value

    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'] + ['referen
        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.

In [10]:
```
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'] + ['referen
        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:

In [11]:
```
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], referen
            })
        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

In [12]:
```
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.

In [13]:
```
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.

In [14]:
```
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.

In [15]:
```
metrics = []
for name, config in configs.items():
    print(f'Evaluating {name!r} model')
    performance = Evaluator().apply(output[name]['prediction'].values, targets[:, 0
    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_valu
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		sensitivity	
subset	train	test	train	test	train	test	train	test	train	test
model										
decision_tree_all_features	8045	2591	36	198	6806	2224	113	87	1.00	0.93
decision_tree_selected_features	8058	2585	23	204	6809	2197	110	114	1.00	0.93
svm_all_features	7887	2709	194	80	6282	2227	637	84	0.98	0.97
svm_selected_features	7890	2715	191	74	6290	2235	629	76	0.98	0.97
bayes_all_features	7064	2501	1017	288	4826	1948	2093	363	0.87	0.90
bayes_selected_features	7009	2380	1072	409	4792	2002	2127	309	0.87	0.85

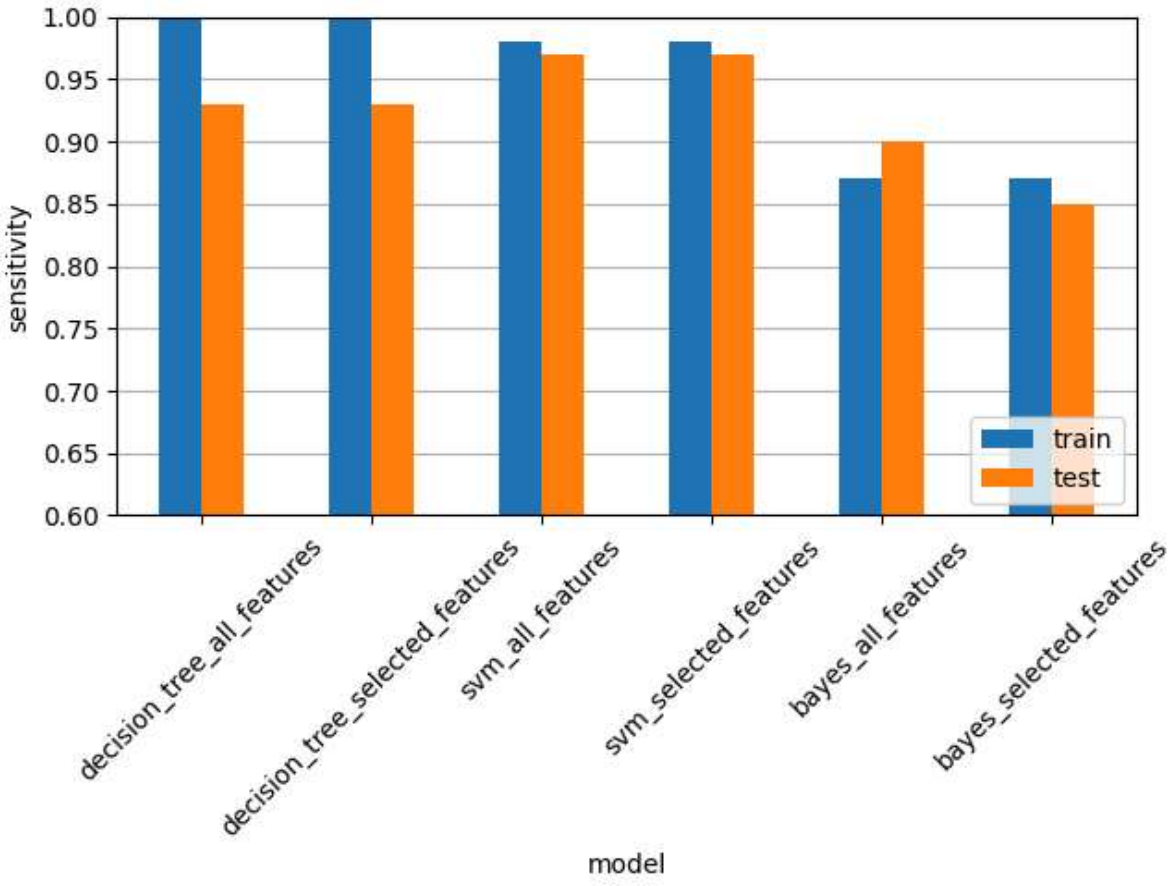
We can also plot the different metrics.

In [16]:
```
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')
```

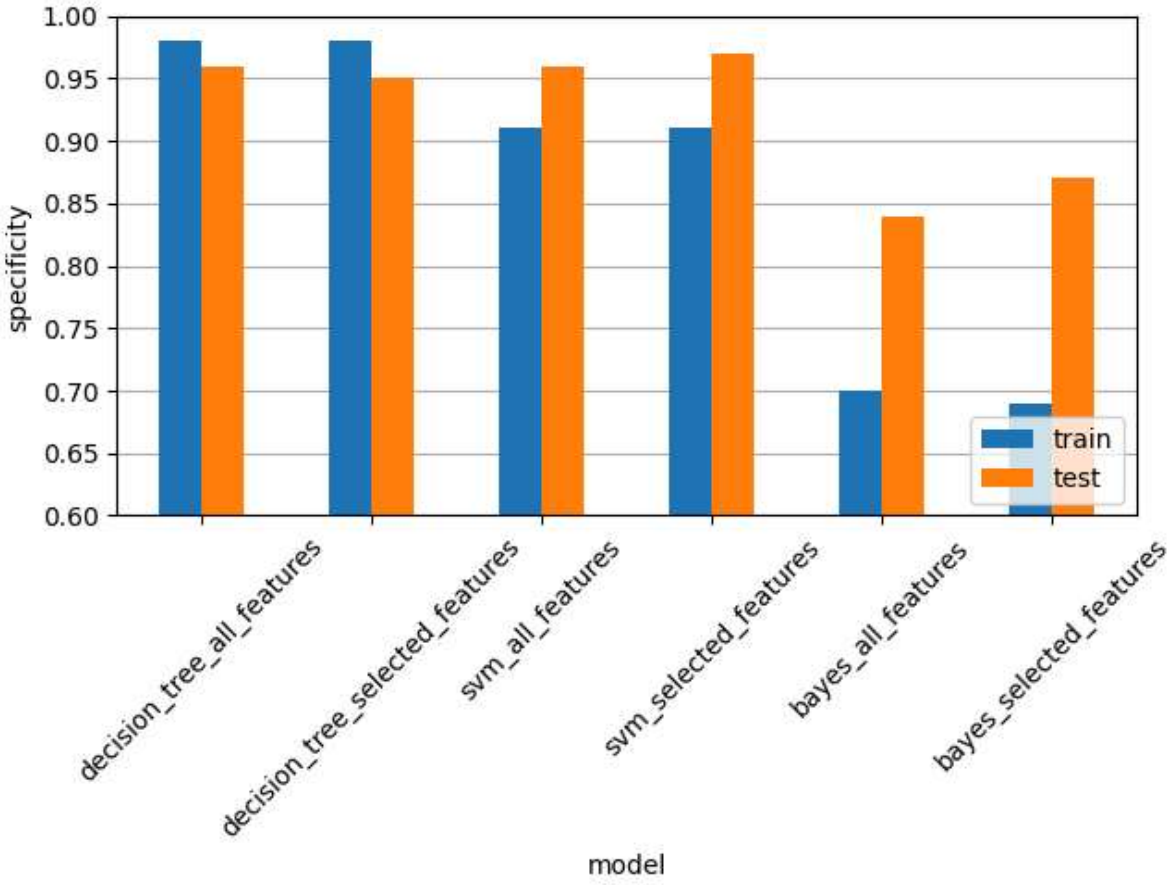
```
        plt.gca().xaxis.set_tick_params(rotation=45)


plot_metrics(metrics)
```

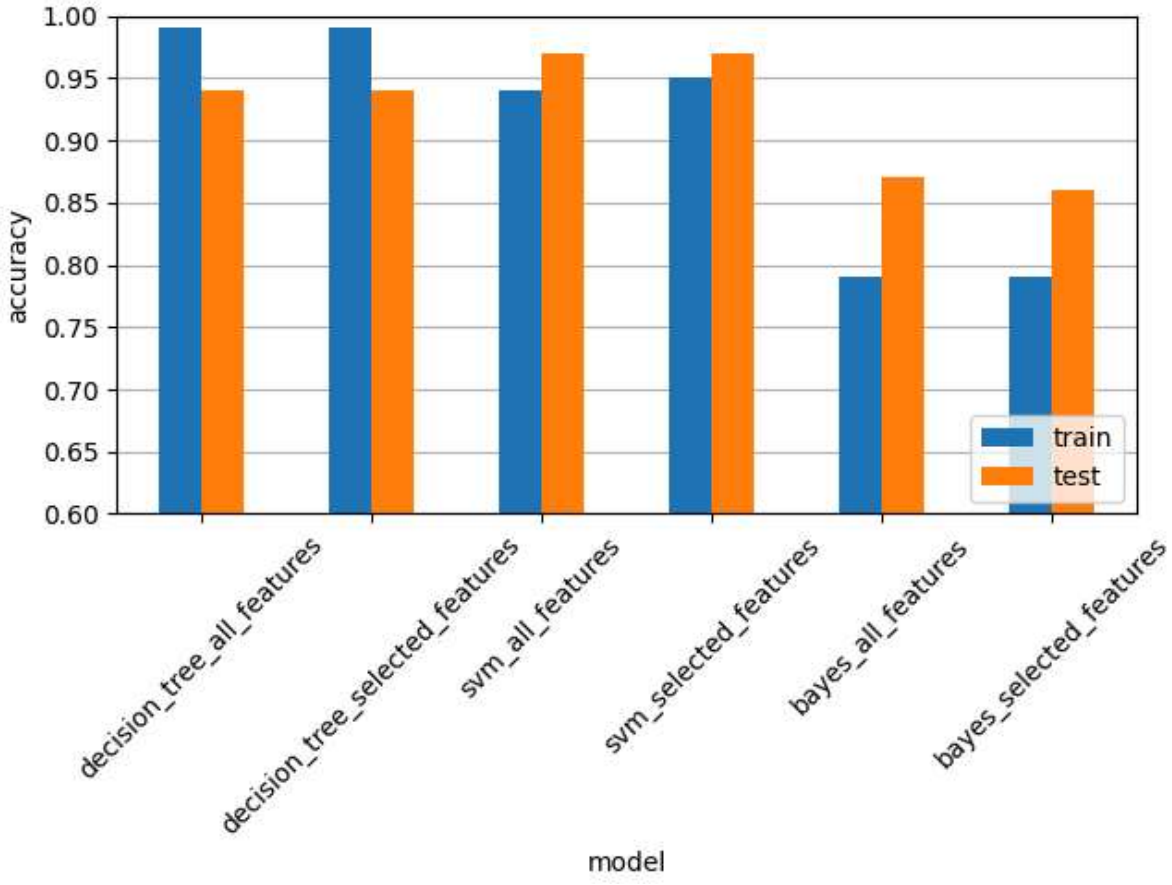
Figure



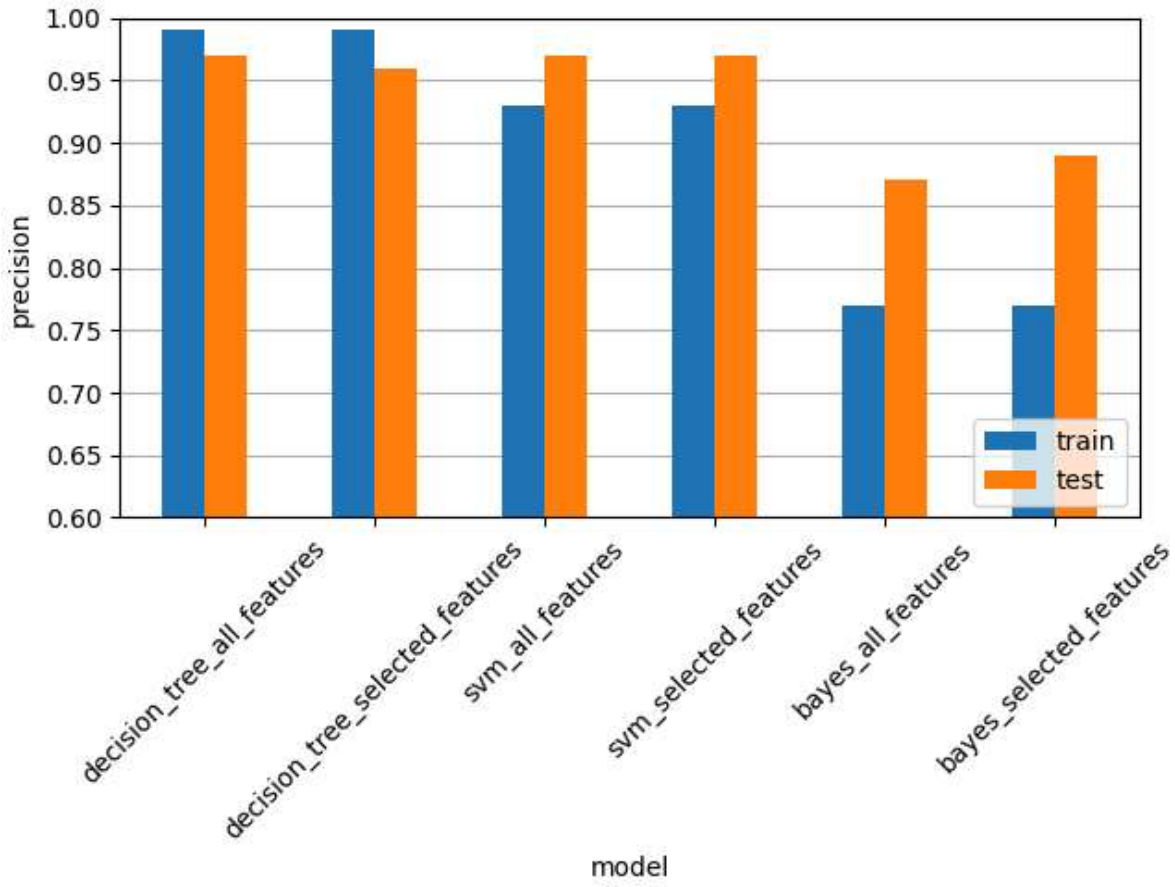
Figure



Figure



Figure



P6. If you want to select a set of features manually, which features would you choose and why?

- The most necessary features tha we would select would be the mean/median of the RR intervals, which capture the capture the baseline heart rate, as well as the std/variance and peak-to-peak (max−min) distance which quantifies the beat-to-beat irregularity that is characteristic of AF. In fact, the scatter/pair plots show that AF segments have higher dispersion and more extreme values. Therefore, we should also keep an extreme such as min or max to catch very short/long intervals.

- More concretely, we could choose `mean` and/or `median_2`, which are the central tendencies, and `std` and/or `var`, as well as `p2p`. We should also select `min` and `max`.

P7. Do the automatically selected features match your manually selected features? Explain the reasons for any similarities and/or differences.

- Mostly yes, the selector kept `median_2`, `p2p`, `mean`, `std`, and `min`, which aligns with the same central-tendency and dispersion cues we would pick. It dropped redundant features (e.g., variance vs std, max vs p2p) because they provide overlapping information, so it converged to a compact, non-collinear subset.

- The main differences are that the selector prefers the transformed `median_2` instead of the raw median and removes strongly correlated features like `var`, `max` and `mean_3`, which is expected from a lasso-based method that penalizes redundant coefficients.

P8. Do you see any signs of overfitting and/or underfitting of the models? Why?

- The decision trees show a bit of overfitting: train accuracy ≈0.99 vs test ≈0.94 with a sensitivity drop from 1.00 to ~0.93. The SVMs generalize well (train ≈0.945, test ≈0.97) with only small gaps, suggesting little over/underfitting. Naive Bayes underfits: both train and test accuracies are low (~0.79/0.86) because its feature-independence assumption limits capacity.

P9. Considering all conditions, which model will you finally choose to detect atrial fibrillation? Why?

- We would choose the SVM with selected features: it delivers the best balanced test metrics (accuracy ≈0.97, sensitivity ≈0.97, specificity ≈0.97, precision ≈0.97), matches the all-features SVM while using fewer features, and has small train–test gaps, so it is robust and efficient for deployment.

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 need to install the required python packages as follows:

In [ ]:
```
# pip install -r "requirements.txt"
```

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

In [ ]:
```
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
%matplotlib widget


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.

In [2]:
```
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 hea
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.

In [3]:
```
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,
                 noverlap=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])
```

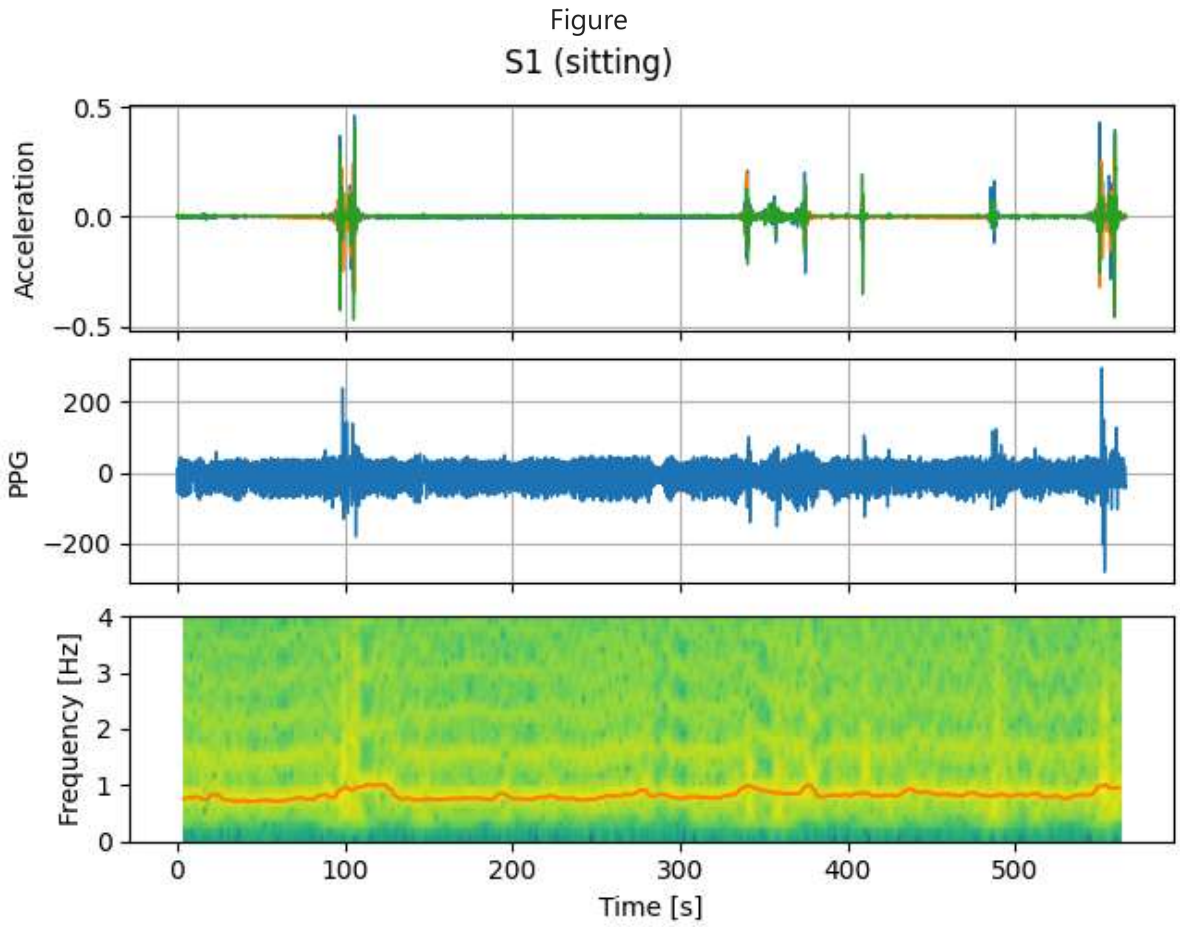
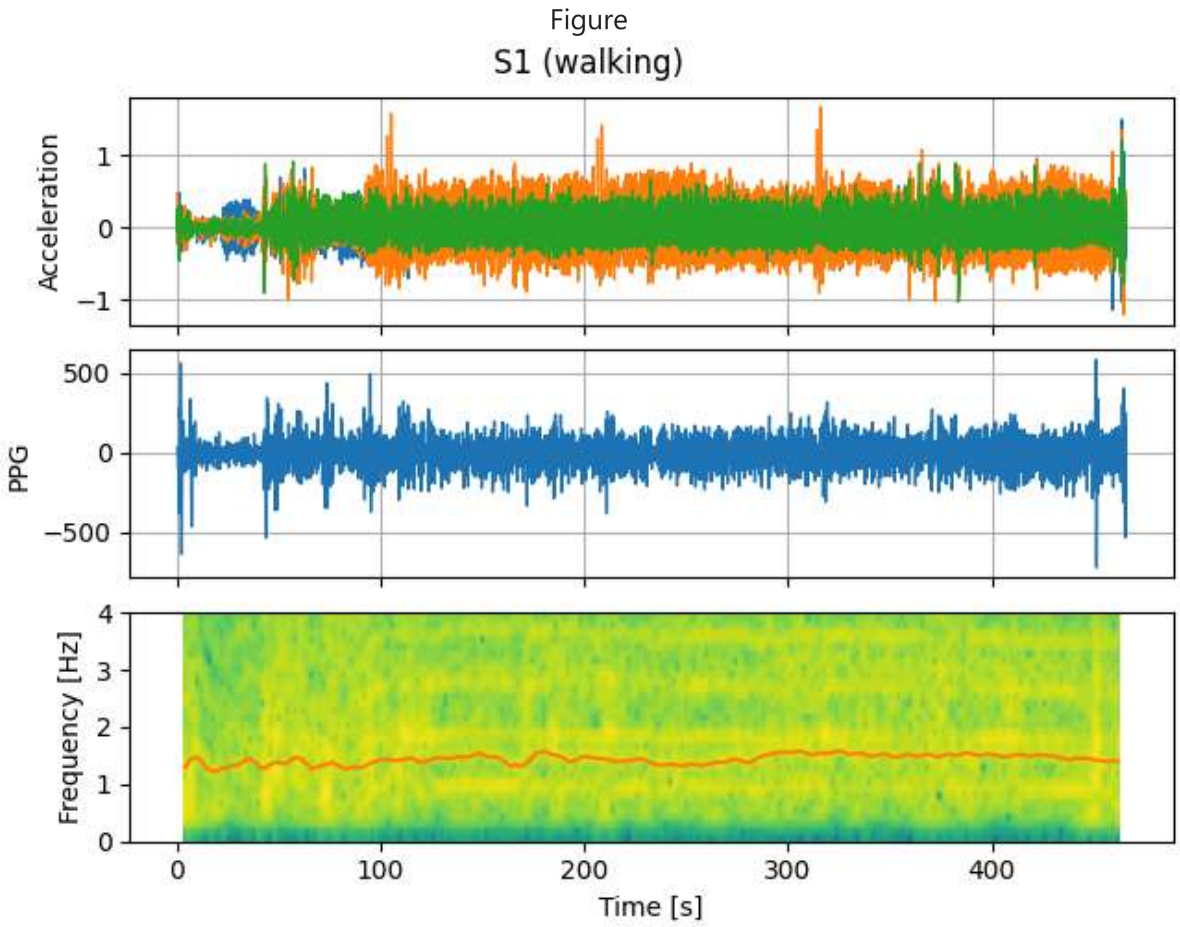


Figure
S1 (sitting)
Acceleration
0.5
0.0
−0.5
PPG
200
0
−200
Frequency [Hz]
4
3
2
1
0
0
100
200
300
400
500
Time [s]



By zooming on the PPG signal, it is clear that walking causes 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).

In [4]:
```python
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 accleration windows: {ppg_acc_windows.shape}')
print(f'Shape of PPG windows: {ppg_windows.shape}')
```

```
Shape of PPG and accleration 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.

In [5]:
```
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]))

    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.

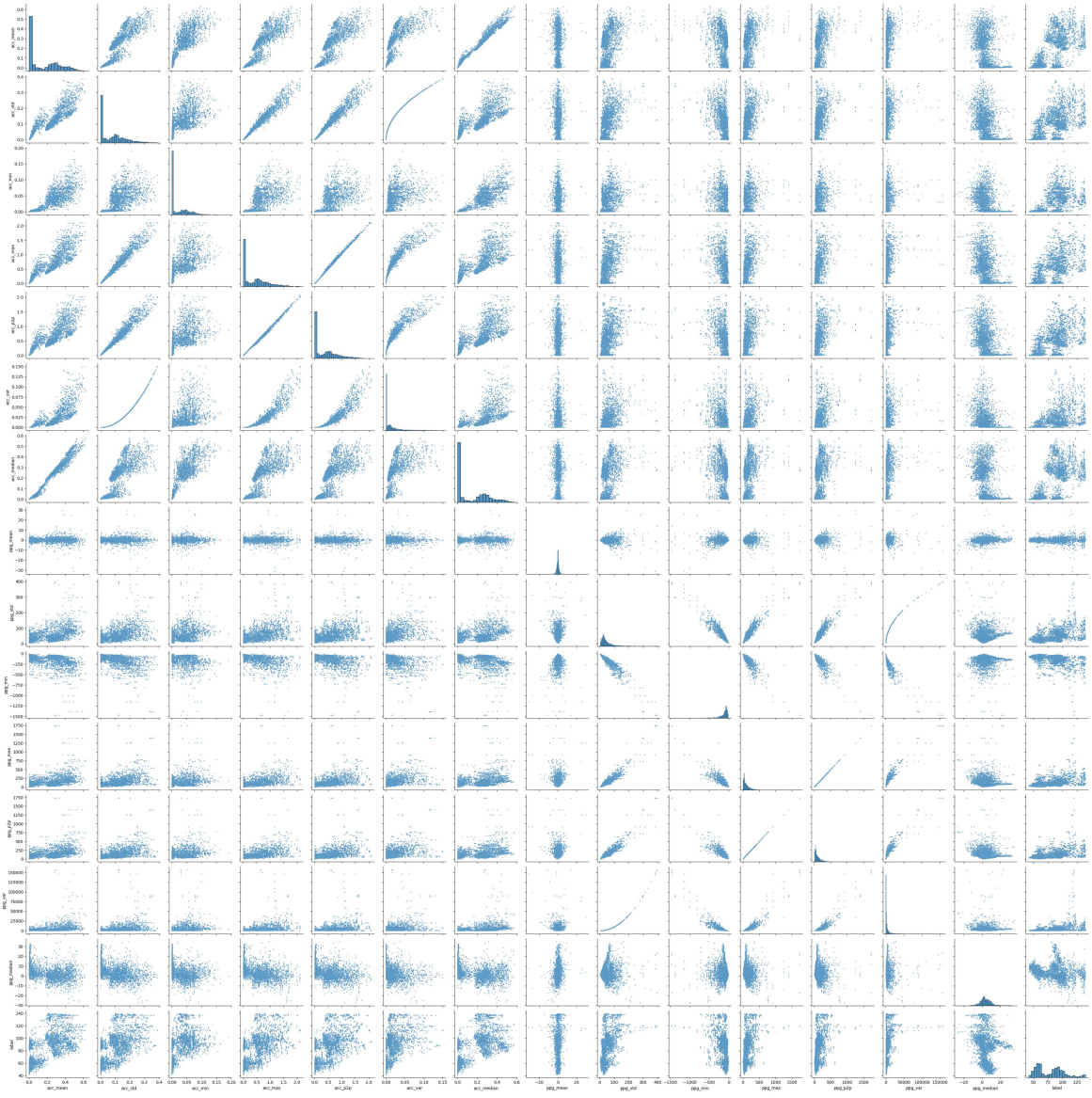
In [6]:
```
acc_norm = np.zeros([ppg_acc_windows.shape[0], ppg_acc_windows.shape[2]])
for instance in range(ppg_acc_windows.shape[0]):
    ppg_acc = ppg_acc_windows[instance, 1:, :]
    acc_norm[instance, :] = np.sqrt(np.sum(np.power(ppg_acc, 2), axis=0))


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(acc_norm, 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])
```

Figure



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

In [7]:
```
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=F
        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
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.

In [8]:

```
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.

In [9]:
```
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'].value

    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'] + ['referen
        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.

In [10]:
```
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'] + ['referen
        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:

In [11]:
```
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[in
            })
        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

In [12]:
```
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.

In [13]:
```
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.

In [14]:
```
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.

In [15]:
```
metrics = []
for name, config in configs.items():
    print(f'Evaluating {name!r} model')
    performance = Evaluator().apply(output[name]['prediction'].values, targets[:, 0
    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_valu
metrics = metrics.unstack().reindex(index=index, columns=columns)
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 ***
```

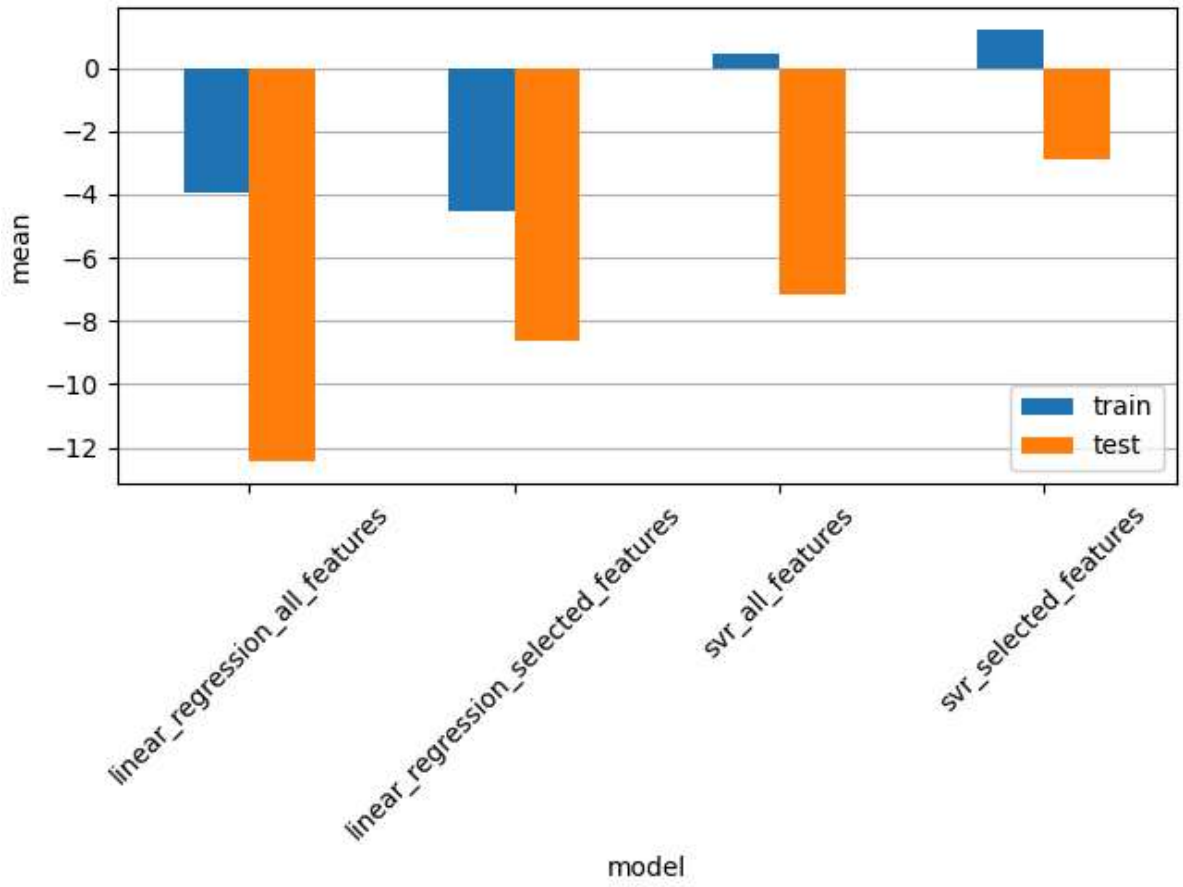
	mean		std			
subset	train	test	train	test	train	
model						
linear_regression_all_features	-3.924053	-12.435107	19.565851	17.199001	19.955468	21
linear_regression_selected_features	-4.549709	-8.650772	20.517729	16.334608	21.016114	18
svr_all_features	0.418621	-7.181589	16.625634	15.572246	16.630904	17
svr_selected_features	1.217230	-2.908962	17.959907	10.697804	18.001109	11

We can also plot the different metrics.

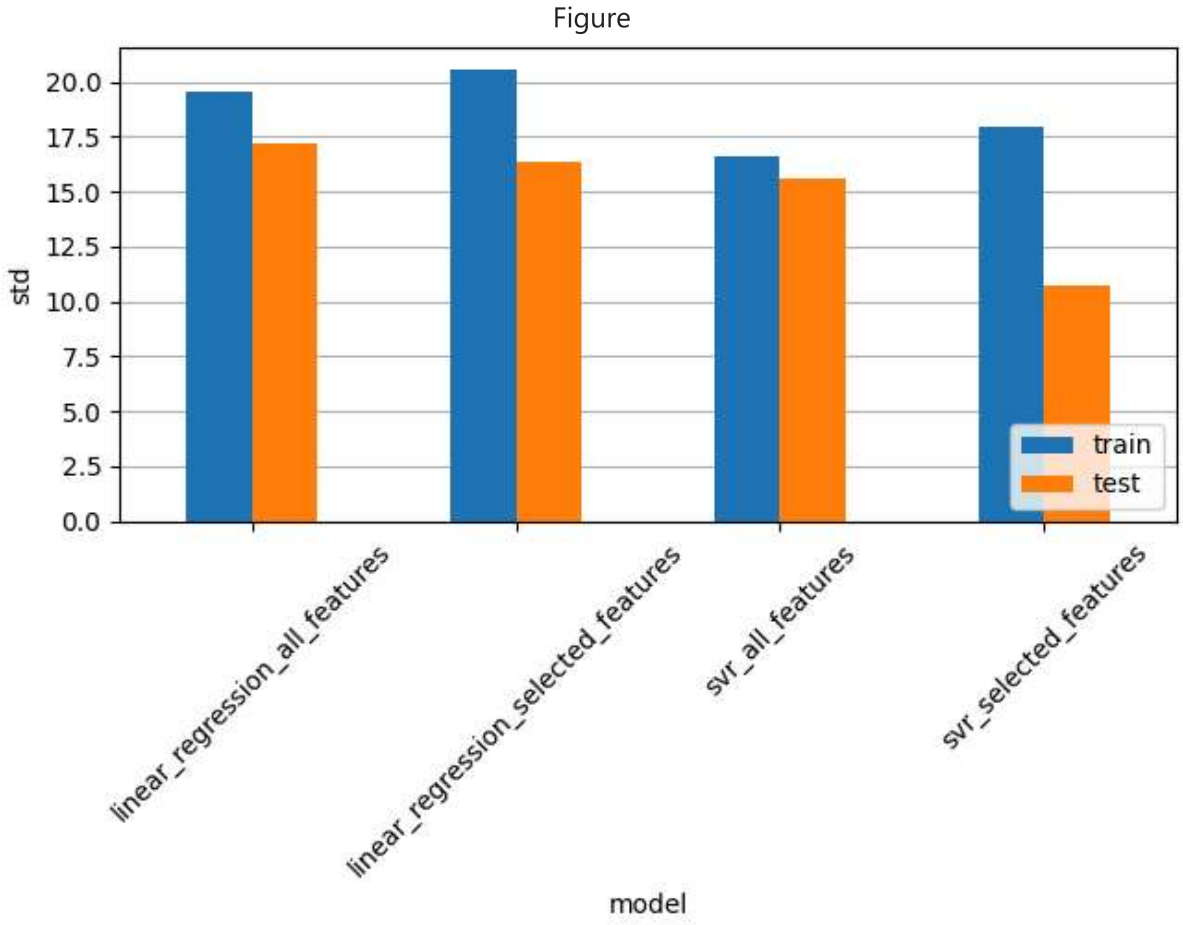
In [16]:
```python
def plot_metrics(data):
    for metric in data.columns.get_level_values(0).unique():
        if metric == 'count':
            continue
        df = data[metric]
        plt.figure(constrained_layout=True)
        plt.gca().set_axisbelow(True)
        df.plot(kind='bar', ylabel=metric, ax=plt.gca())
        plt.grid(axis='y')
        plt.legend(loc='lower right')
        plt.gca().xaxis.set_tick_params(rotation=45)


plot_metrics(metrics)
```

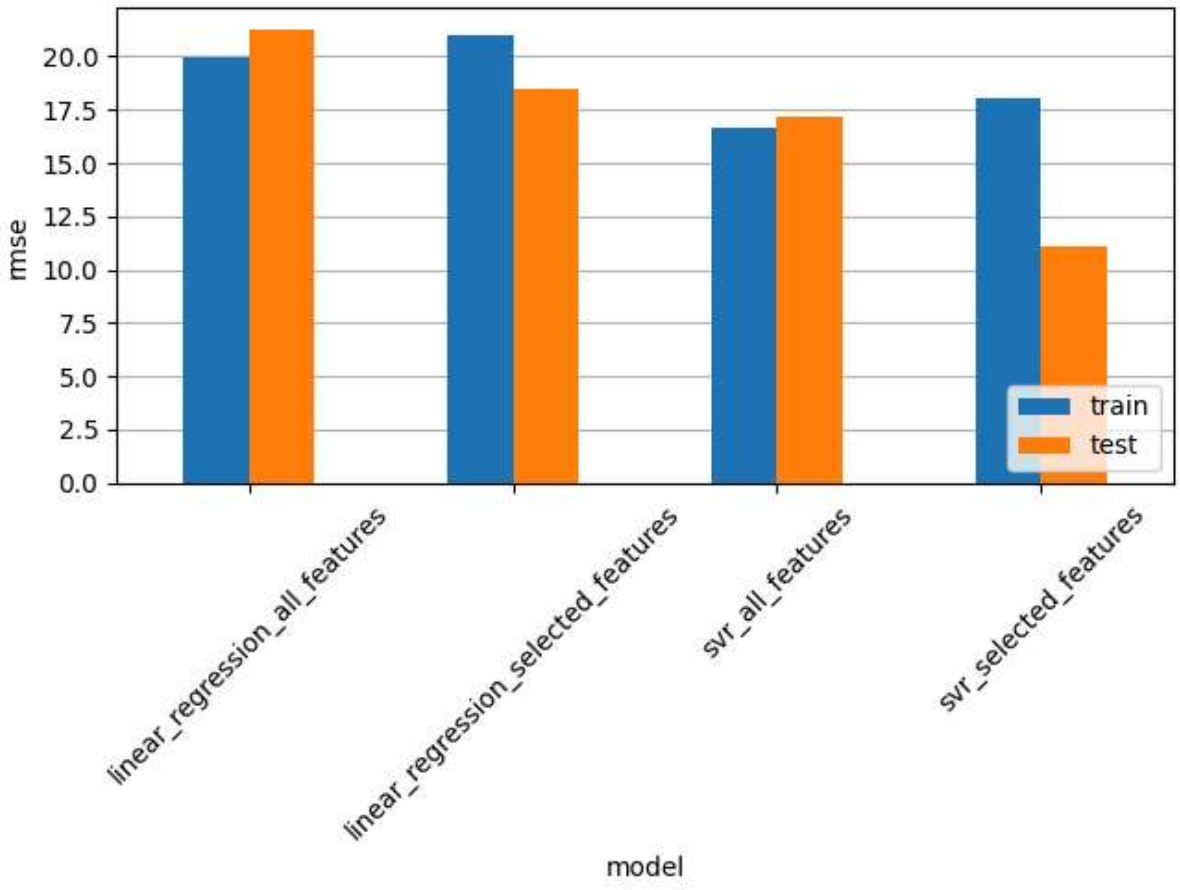
Figure



Figure



Figure



P1. If you want to select a set of features manually, which features would you choose and why?

- From the accelerometer acc_mean, acc_std, acc_min, acc_max, acc_median since they capture movement intensity, variability, and extremes; and from the PPG ppg_mean, ppg_std, ppg_min, ppg_max, ppg_median since they contain important information about PPG amplitude and variability which relates to pulse strength. The selected features summarize central tendency and variability and capture relevant information while avoiding redundant features such as p2p and var.

P2. Do the automatically selected features match your manually selected features? Explain the reasons for any similarities and/or differences.

- The automatic method chose a subset of the manually selected features. Automatic selection chooses features based on their statistical contribution to the target variable, therefore the algorithm may detect subtle correlations or remove features that look meaningful but don't add predictive value.

P3. Do you think that the feature selection was useful in this exercise? Why?

- Yes. Feature selection reduced overfitting. The test errors decreased for both models when using fewer features. It simplified the models by removing redundant or

uninformative features.

P4. How do you interpret mean, std, and rmse errors of the models?

- They describe different aspects of model accuracy. Mean shows bias, std shows variability (consistency) of the models, and rmse shows total error magnitude which is more sensitive to big mistakes and is overall the best metric for comapring the models.

P5. Considering all conditions, which model will you finally choose to estimate heart rate? Why?

- The SVR model with selected features since it has the best overall performance on the test set. It achieves the lowest test RMSE, meaning it generalizes better and gives the most reliable estimates across conditions.