bci4als: a Complete Pipeline for Motor Imagery Classification

Evyatar Luvaton, Noam Siegel

December 28, 2020

Part 1: End-to-End Walkthrough

This software was developed for the course Brain-Computer-Interface for ALS Patients, December 2020. It is available at https://github.com/evyatarluv/BCI-4-ALS

Over the mid-semester project we have integrated the different parts of BCI which been discussed in the course. We introduce bci4als, a complete pipeline for EEG motor imagery data recording and classification.

The pipeline is general, so it can be extended to other BCI settings like P300, SSVEP.

The first part of this report presents the python package bci4als.

Preliminaries

First, go ahead and install bci4als (if you haven't already done so).

```
pip install -i https://test.pypi.org/simple/ bci4als
```

Project Structure

Structure your project exactly as shown:

[1]:
```
print(open('project_structure_0.txt', 'r', encoding='utf-8').read())
```

```
project_name
 data
    subject1
    subject2
 main.py
```

The main codebase lies in the `bci4als.mi` module:

[2]:
```
import bci4als.mi as mi
```

Getting Started

We will now guide you how to use the `bci4als` pipeline from end to end.

Note: All the scripts share common configuration parameters, which are stored in a multi-level `dict` object called `mi.params`. The first-layer keys indicate the category of parameters:

```
[3]: params = mi.params
     params.keys()
```

```
[3]: dict_keys(['gui', 'lsl', 'visual', 'experiment', 'preprocess', 'data',
     'features', 'split'])
```

For example, to see the parameters related to preprocessing:

```
[4]: preprocess_params = params['preprocess']
     print('\n'.join('{}: {}'.format(k, v) for k, v in preprocess_params.items()))
```

```
filter: {'high_pass': 0.5, 'low_pass': 40, 'notch': 50}
channel_names: ['time', 'C03', 'C04', 'P07', 'P08', 'O01', 'O02', 'F07', 'F08',
'F03', 'F04', 'T07', 'T08', 'P03']
remove_channels: [0, 1, 2]
```

Step 1 - Record Experiment

The `bci4als.mi.record` module is responsible for executing a motor imagery experiment and recording the stimulus.

The `bci4als.mi.record.start()` function starts an interactive gui which will guide you through executing a mi experiment. it assumes you:

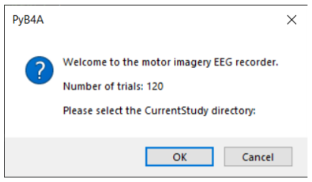
1) have an external application streaming eeg data (via lsl) to lab recorder.
2) have a `currentstudy` directory into which the collected data will be saved.
3) opened `labrecorder`.

Once you are ready, begin recording:

`mi.record.start()`

Now follow the following steps:

1) You should see a "Welcome" messagebox. Press "OK" to continue.


Welcome to the motor imagery EEG recorder.
Number of trials: 120
Please select the CurrentStudy directory:

2) Select the CurrentStudy folder.
3) Type a new Session ID (this will be the name of the folder). Press "OK" to continue.
4) You should a confirmation messagebox. Press "OK" to continue.
5) Point the LabRecorder to the folder created at step 3. Set the name of the file EEG.xdf
6) Make sure you are prepared to begin the motor imagery experiment. Press "OK" to begin the trials.

7) After the experiment, press "OK" to close the program.

Lab Recorder will generate the `EEG.xdf` file which contains the EEG and markers recording. It can be read with pyxdf.

Python will generate `stimulus_vector.csv`, which contains the labels for each trial.

Your project structure should look like this:

[5]:
```
print(open('project_structure_1.txt', 'r', encoding='utf-8').read())
```

```
project_name
 data
    subject1
    subject2
           session1
                EEG.xdf
                stimulus_vectors.csv
 main.py
```

Step 2 - Data Preprocessing

The `preprocess.py` script is responsible for cleaning the data.

Currently, the pre-processing script uses low-pass, high-pass and notch filters to clean the data.

The parameters for the filters are also part of the `params` object:

[6]:
```
params['preprocess']['filter']
```

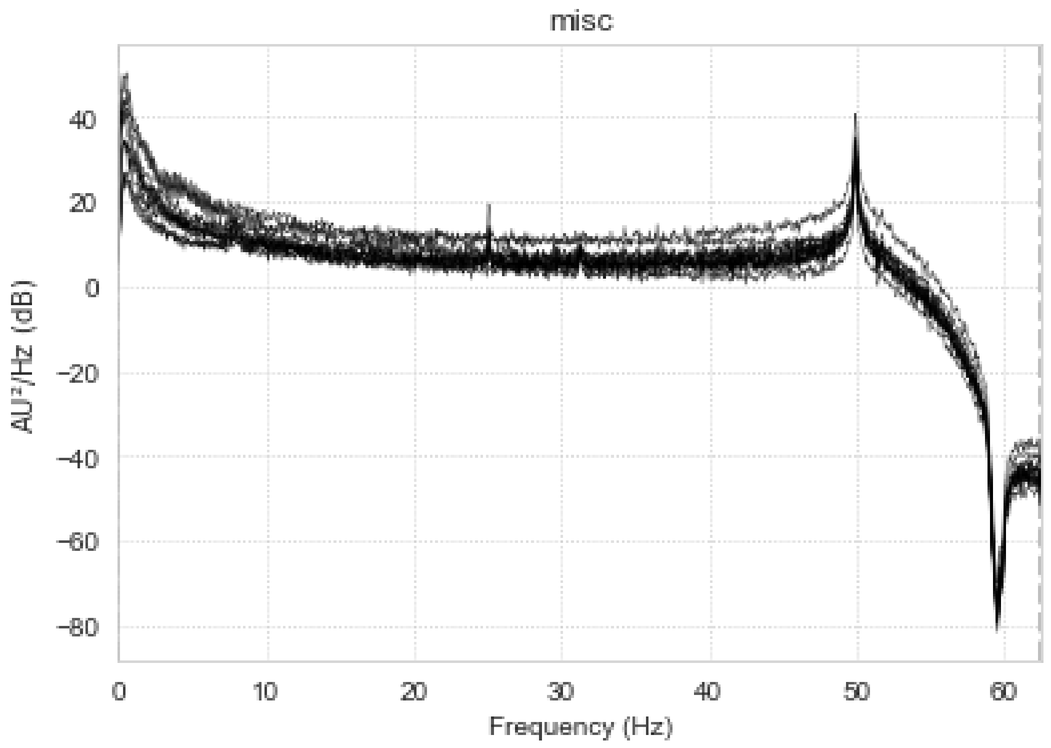
[6]:
```
{'high_pass': 0.5, 'low_pass': 40, 'notch': 50}
```

The script will output an `EEG_clean.csv` file, which contains the EEG data after the pre-processing. Additionally, the script exports a `.info` json file. The `.info` file contains all the info about the EEG stream.

Let's plot the clean EEG data:

[19]:
```
# Params
eeg_clean_path = '../data/noam/2/EEG_clean.csv'
ch_names = ['C03', 'C04', 'P07', 'P089', 'O01', 'O02', 'F07', 'F08', 'F03',␣
 ↪'F04', 'T07', 'T08', 'P03']
s_rate = 125

# Create mne info & raw
eeg_clean = np.genfromtxt(eeg_clean_path, delimiter=',', skip_header=1)[:, 1:]
info = mne.create_info(ch_names, s_rate, verbose=False)
raw_eeg_clean = mne.io.RawArray(eeg_clean.T, info, verbose=False)

# Plot
```

```
fig = raw_eeg_clean.plot_psd(picks=ch_names, show=False)
```



Step 3 - Data Segmentation

In the `segment_data.py` script we split the data for each trial. The start and end of each trial is according to the markers streaming we created while recording the EEG. The script gets the subject folder from the config file (under `config['data']['subject_folder']`) and splits for trial each EEG record in each day.

The output of MI3 script is a pickle file named `EEG_trials.pickle` which is located in the corresponding session directory.

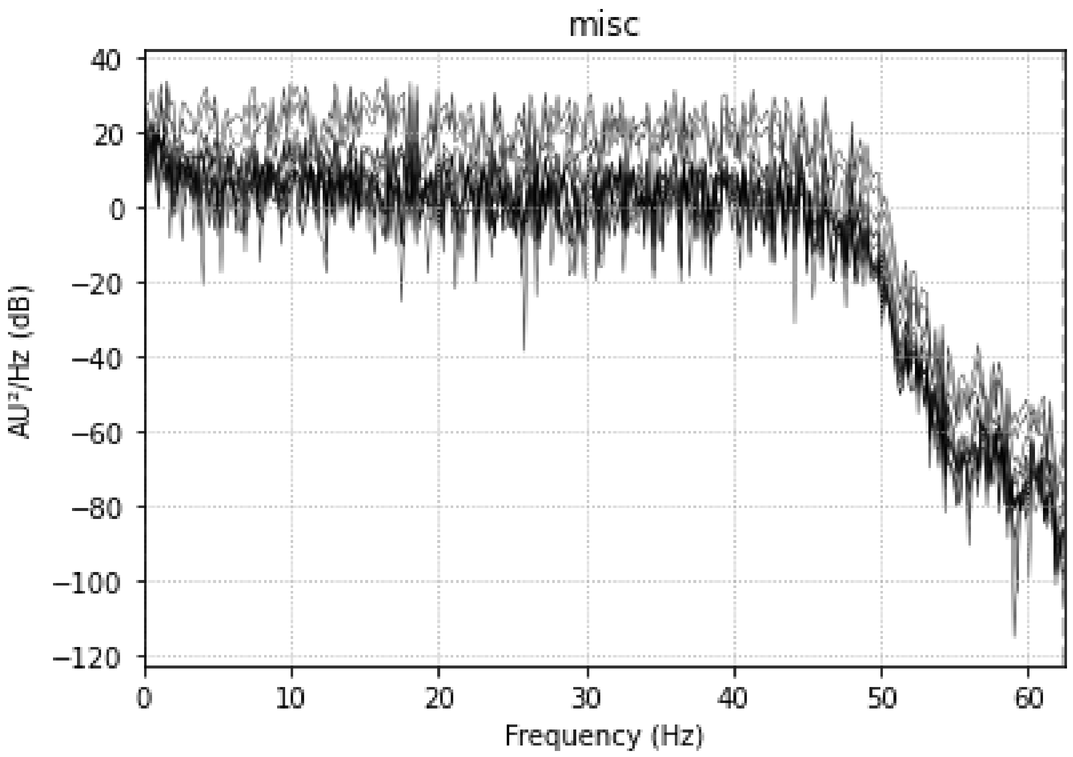
The file is a list of `ndarrays`, where each element is the corresponding EEG data of the trial.

[10]:
```
eeg_trials_path = '../data/noam/3/EEG_trials.pickle'
eeg_trials = pickle.load(open(eeg_trials_path, 'rb'))

print('Number of trials: {}\nTrials dimensions: {}'.format(len(eeg_trials),␣
 ↪eeg_trials[0].shape))
```

```
Number of trials: 120
Trials dimensions: (616, 13)
```

And a PSD plot of a specific trial looks like:

```
[11]: eeg_trial = mne.io.RawArray(eeg_trials[103].T, info, verbose=False)
      fig = eeg_trial.plot_psd(picks=ch_names)
```



Step 4 - Feature Extraction

We attempt to extract statistical features which will enable the classifiers to differentiate between the Left, Right and Idle mental states.

Selected Channels Motor imagery and motor execution are assumed to share a similar underlying neural system that involves primary motor cortex. That is why we select channels C3 and C4, which are located over the motor cortex:

```
[12]: params['features']['selected_channels']
```

```
[12]: ['C03', 'C04']
```

Selected Features We leverage the mne-features api. The api provides 27 different feature types which can be extracted. The `config.yaml` configuration file allows easy control of the feature selection.

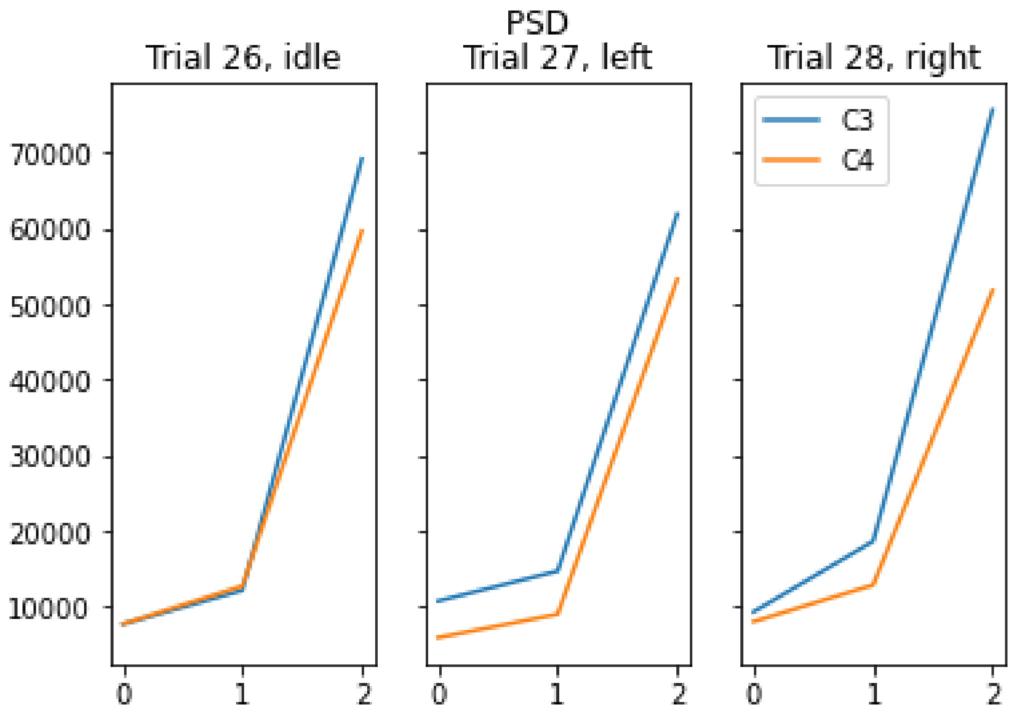
We can see the currently selected features:

```
[13]: params['features']['selected_features']
```

```
[13]: ['energy_freq_bands']
```

chose to extract the powerband energies in the ranges Low Alpha (8-10Hz), High Alpha (10-12.5 Hz) and Beta (12.5-30 Hz) for each electrode. Thus, in total we have 6 features per trial.

Here is a sample of the features extracted:



Step 5 - Train and Test a Model

The next script concentrates on training and testing the model. The main function in this module is the `train()` function. The function gets 2 arguments - `mode` and `model_name`.

The `mode` argument decide how to treat the different days in the subject folder. Because the data was recorded on different days with different environmental variables, we expect it to be different. We can't train a model on the first day and expect for good results on the other days. So we created 3 different modes for splitting the data into train and test:

1. `same_day` - on the `same_day` mode the data will be split for each day separately. Each day will have his own train and test data according to the train ratio which can be found in the config file. Hence, the model will be trained on the data from the current day and also will be tested on it.

2. `first_day` - on this procedure the model will be trained only on the first day and will be tested on the other days.

3. `adjust` - on the adjust mode the model will be trained each day on the first day with some more data samples from the current day. In this mode we use previous data in each day so there is no need to train model from scratch.

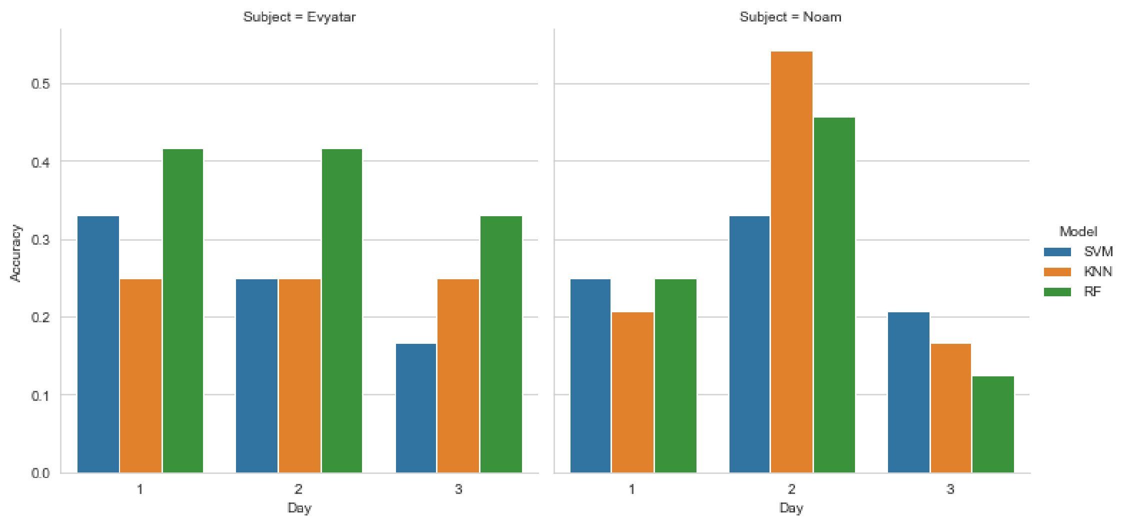
The second argument in the function, i.e., `model_name` represent the name of the ML model which will be used to classify. Currently we implemented SVM, KNN and Random Forest models, but every developer can easily add his desirable model to the function.

An example for using the `train()` function is:

```
mi.assess_model.train(mode='same day', model='svm')
```

Results We trained different type of models and tested them on our EEG data. We used only `same_day` mode in the `train()` function. The results can be seen here:

```
[15]: results = pd.read_csv('results.csv')
      sns.set_style(style='whitegrid')
      sns.catplot(x='Day', y='Accuracy', hue='Model', col='Subject', data=results,␣
       ↪kind='bar')
      plt.show()
```



Unfortunately, the models fails to perform well on the test set. We believe that there are two main reasons for this:

Artefacts

The raw data could be contaminated by unwanted sources. We tried to remove technical disturbances as much as possible, for example unplugging the computer chargers and digitally blocking the 50 Hz frequency. There could still be biological artefacts from the subject, such as eye movements, muscle activities, tongue movements and sweating that have not been controlled.

Feature Complexity

The neuronal dynamics which are responsible for the oscillations detected by EEG signals are highly complex and nonlinear. In order to be able to classify different data points, the combination of the feature extraction methods and the classification model needs to be adapted to the problem at hand. It is possible that a different selection of features or classification methods would lead to better results.

Part 2: Experiment with ResNet Features

Pursuant to the instructions, we implemented a different approach for one of the above-mentioned steps. We chose to re-implement the features extraction step. Over the first time we used a classic approach in order to extract features from the EEG data to the ML model. This classic approach required domain knowledge in the data - which channels are more important for MI classification, more common features for the classification problem, etc. In the re-implementation part we want to implement feature extraction method which do not demand any domain-knowledge, and to test the result of the model. We were constructed not to use any deep learning algorithm for the model step, but hey, we can definitely use one for the feature extraction step!

We decided to use pre-trained Convolutional Neural Networks (CNNs) in order to extract features from each EEG trial. The CNN consist of convolutional layers followed by fully-connected layers. So, we used the pre-trained ResNet, which been trained on the 'imagenet' dataset, and used it where `include_top=False`, i.e., we used only the output of the convolutional layers. The ResNet input image size needs to be at list `(32, 32)`. Since we got only 15 channels we needed to resize the EEG data in order to fit the ResNet input shape. The initial EEG resize is the first hyperparameter.

Using a pooling layer, we got from the ResNet a vector with 2048 features for each EEG trial. Then we used a PCA algorithm for dimensionality reduction. The number of components using the PCA is our second hyperparameter. Now we used it as our feature vector for the ML model.

All the hyperparameters we used can be found in the `config.yaml` configuration file.

Now we used our `extract()` function to extract features using ResNet:

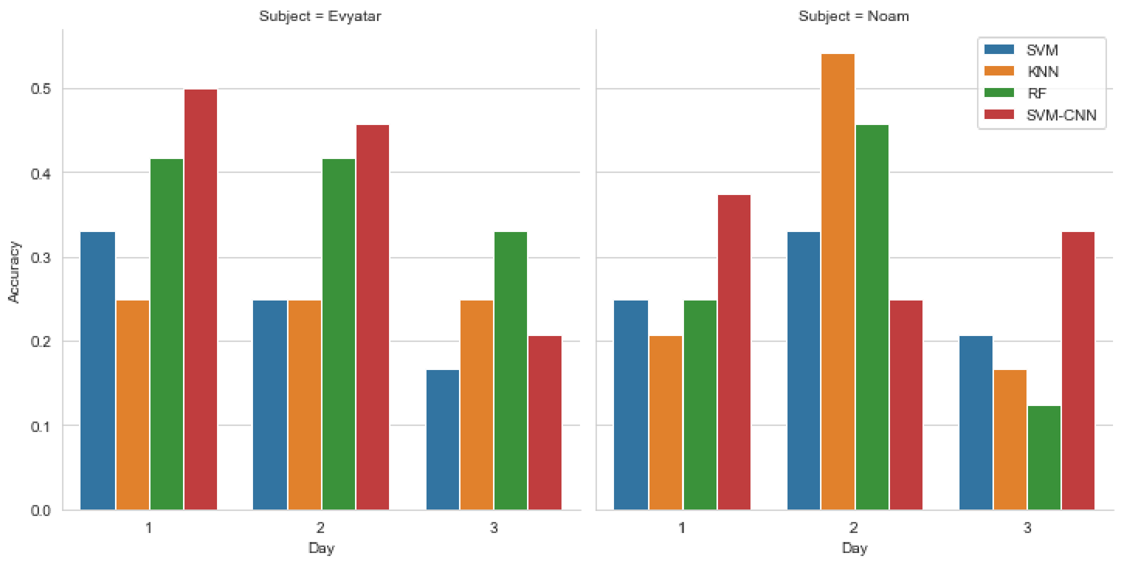
```
mi.extract_features.extract(mode='cnn')
```

And the full pipeline using ResNet as feature extractor should look like:

```
# Assuming you already record an experiment

mi.preprocess.preprocess()

mi.segment_data.segment_data()

mi.extract_features.extract(mode='cnn')

mi.assess_model.train('same day', 'svm')
```

Now let's look at the results of the ResNet performance using SVM for classification:

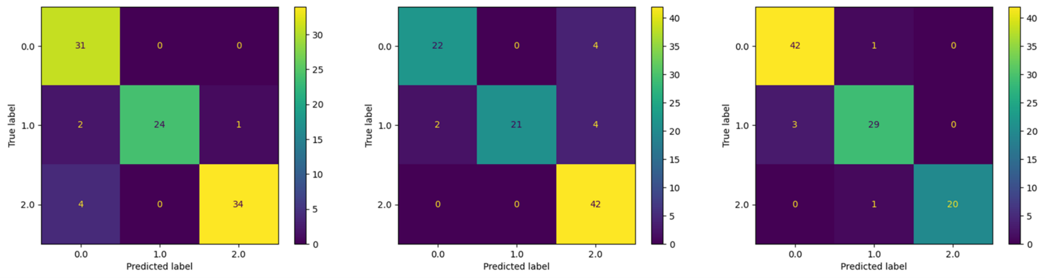
[16]:
```
results = pd.read_csv('results_cnn.csv')
sns.set_style(style='whitegrid')
sns.catplot(x='Day', y='Accuracy', hue='Model', col='Subject', data=results,␣
 ↪kind='bar', legend=False)
plt.legend(bbox_to_anchor=(1, 1))
plt.show()
```



Alas, the results are still bad, even with the CNN-extracted features. It is interesting to look at the confusion matrix of the SVM-CNN classifier on one of the subjects, across three consecutive days. First on the training set, then on the test set:

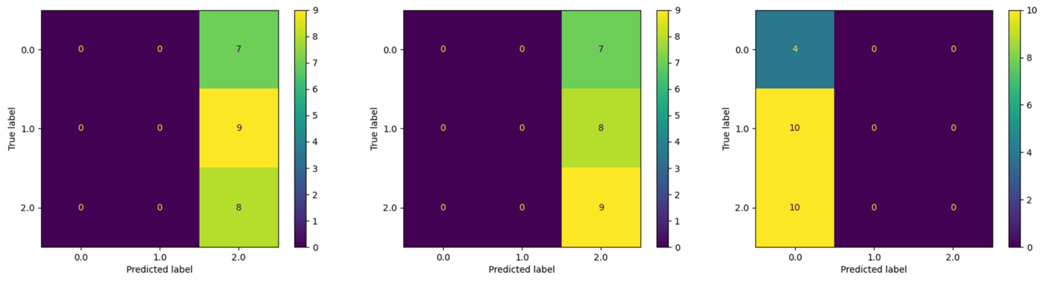
[17]:
```
Image(filename='images/confusion_matrix_train.png')
```

[17]:



[18]:
```
Image(filename='images/confusion_matrix_test.png')
```

[18]:



We note that:

1. The classifier performs well on the training set (sanity check).
2. The classifier predicts one label across the entire test set.
3. The label which is predicted is the one with the most samples in the training set.

This is a possible indication that the feature space is too large compared to the dataset set. In that case, the points lie too close together in the large space for the svm to differentiate. Then it will guess the label which was observed most frequently in the training set.