In [1]:
```
# ECG Data Analysis & Classification
```

In [ ]:
```
# The Heartbeat types include 5 categories which are N: normal, S: supraventricular, V: ventricular, F: fusion, and Q:
# They are represented by the numbers 0.0, 1.0, 2.0, 3.0, 4.0 separately
```

In [ ]:
```
# Part 1: Data Preparation
```

In [1]:
```
import Pkg; Pkg.add("DataFrames")
import Pkg; Pkg.add("CSV")
using CSV, DataFrames, Plots
```

```
    Updating registry at `C:\Users\zizhe\.julia\registries\General.toml`
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
```

In [2]:
```
cd("C:/Users/zizhe/Desktop/411 datasets/archive")
```

```
IOError: cd("C:/Users/zizhe/Desktop/411 datasets/archive"): no such file or directory (ENOENT)

Stacktrace:
 [1] uv_error
   @ .\libuv.jl:97 [inlined]
 [2] cd(dir::String)
   @ Base.Filesystem .\file.jl:89
 [3] top-level scope
   @ In[2]:1
 [4] eval
   @ .\boot.jl:373 [inlined]
 [5] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String, filename::String)
   @ Base .\loading.jl:1196
```

In [4]:
```
# load the training data

dftr = CSV.read("mitbih_train.csv", header=false,DataFrame)
```

Out[4]: 87,554 rows × 188 columns (omitted printing of 180 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	0.977941	0.926471	0.681373	0.245098	0.154412	0.191176	0.151961	0.0857843
2	0.960114	0.863248	0.461538	0.196581	0.0940171	0.125356	0.0997151	0.0883191
3	1.0	0.659459	0.186486	0.0702703	0.0702703	0.0594595	0.0567568	0.0432432
4	0.925414	0.665746	0.541436	0.276243	0.196133	0.0773481	0.0718232	0.0607735
5	0.967136	1.0	0.830986	0.586854	0.356808	0.248826	0.14554	0.0892019
6	0.927461	1.0	0.626943	0.193437	0.0949914	0.0725389	0.0431779	0.0535406
7	0.423611	0.791667	1.0	0.256944	0.0	0.277778	0.465278	0.520833
8	0.716814	0.539823	0.283186	0.129794	0.0648968	0.0766962	0.0265487	0.0324484
9	0.874214	0.849057	0.480084	0.0587002	0.0901468	0.310273	0.387841	0.385744
10	1.0	0.996086	0.694716	0.336595	0.238748	0.268102	0.191781	0.174168
11	0.985507	0.880435	0.518116	0.213768	0.126812	0.119565	0.0833333	0.0797101
12	1.0	0.657563	0.178571	0.178571	0.136555	0.0357143	0.0210084	0.0672269
13	0.986737	0.962865	0.809019	0.35809	0.0822281	0.161804	0.137931	0.0795756
14	1.0	0.912651	0.593373	0.0451807	0.0361446	0.177711	0.316265	0.343374
15	1.0	0.477157	0.0609137	0.0736041	0.0888325	0.0431472	0.035533	0.0279188
16	0.985632	0.78592	0.359195	0.0962644	0.0431034	0.145115	0.196839	0.175287
17	1.0	0.872727	0.358442	0.0571429	0.0649351	0.0909091	0.0649351	0.0493506
18	0.986072	0.506964	0.350975	0.214485	0.133705	0.097493	0.0835655	0.091922
19	1.0	0.93501	0.622642	0.291405	0.236897	0.224319	0.157233	0.140461
20	0.979644	0.974555	0.765903	0.315522	0.0737913	0.160305	0.145038	0.0687023
21	0.950192	0.777778	0.398467	0.0	0.0	0.118774	0.137931	0.229885
22	1.0	0.920245	0.680982	0.466258	0.257669	0.104294	0.0858896	0.110429
23	0.813754	0.719198	0.398281	0.0	0.0974212	0.237822	0.303725	0.340974

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.997118	0.786744	0.0720461	0.092219	0.0835735	0.0893372	0.0691643	0.0835735
25	1.0	0.536341	0.0	0.0626566	0.0626566	0.0651629	0.0451128	0.047619
26	0.993289	0.825503	0.422819	0.154362	0.104027	0.11745	0.0771812	0.0704698
27	0.961694	0.747984	0.135081	0.00201613	0.0725806	0.0725806	0.0625	0.0604839
28	0.989858	0.906694	0.726166	0.553753	0.359026	0.174442	0.0527383	0.0
29	0.975881	0.923933	0.549165	0.230056	0.187384	0.172542	0.116883	0.115028
30	0.962662	0.928571	0.441558	0.0	0.0519481	0.176948	0.181818	0.206169
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [5]:

```
# Unpack the training dataset to Xtrdf, ytrdf

using MLJ

ytrdf, Xtrdf = unpack(dftr, ==(:Column188),name->true)

# Equivalent to Xtrdf=dftr[:,1:end-1], plus, ytrdf=dftr[:,end],
```

Out[5]: ([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0 … 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], **87554×187 DataFrame**

Row	Column1	Column2	Column3	Column4	Column5	Column6	Column ⋯
	Float64	Float64	Float64	Float64	Float64	Float64	Float6 ⋯
1	0.977941	0.926471	0.681373	0.245098	0.154412	0.191176	0.1519 ⋯
2	0.960114	0.863248	0.461538	0.196581	0.0940171	0.125356	0.0997
3	1.0	0.659459	0.186486	0.0702703	0.0702703	0.0594595	0.0567
4	0.925414	0.665746	0.541436	0.276243	0.196133	0.0773481	0.0718
5	0.967136	1.0	0.830986	0.586854	0.356808	0.248826	0.1455 ⋯
6	0.927461	1.0	0.626943	0.193437	0.0949914	0.0725389	0.0431
7	0.423611	0.791667	1.0	0.256944	0.0	0.277778	0.4652
8	0.716814	0.539823	0.283186	0.129794	0.0648968	0.0766962	0.0265
9	0.874214	0.849057	0.480084	0.0587002	0.0901468	0.310273	0.3878 ⋯
10	1.0	0.996086	0.694716	0.336595	0.238748	0.268102	0.1917
11	0.985507	0.880435	0.518116	0.213768	0.126812	0.119565	0.0833
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮ ⋱
87545	0.621951	0.521341	0.463415	0.457317	0.417683	0.362805	0.2957
87546	0.799242	0.515152	0.545455	0.55303	0.526515	0.503788	0.4242 ⋯
87547	0.757235	0.663987	0.561093	0.453376	0.324759	0.21865	0.1446
87548	0.717325	0.62766	0.534954	0.427052	0.322188	0.215805	0.1367
87549	1.0	0.405594	0.440559	0.405594	0.405594	0.384615	0.3741
87550	0.807018	0.494737	0.536842	0.529825	0.491228	0.484211	0.4561 ⋯
87551	0.718333	0.605	0.486667	0.361667	0.231667	0.12	0.0516
87552	0.906122	0.62449	0.595918	0.57551	0.530612	0.481633	0.4448
87553	0.858228	0.64557	0.84557	0.248101	0.167089	0.131646	0.1215
87554	0.901506	0.845886	0.800695	0.748552	0.687138	0.599073	0.5121 ⋯

181 columns and 87533 rows omitted, **0×0 DataFrame**)

In [6]:
```
# Show all the features (explanatory variable)

Xtrdf
```

Out[6]: 87,554 rows × 187 columns (omitted printing of 179 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	0.977941	0.926471	0.681373	0.245098	0.154412	0.191176	0.151961	0.0857843
2	0.960114	0.863248	0.461538	0.196581	0.0940171	0.125356	0.0997151	0.0883191
3	1.0	0.659459	0.186486	0.0702703	0.0702703	0.0594595	0.0567568	0.0432432
4	0.925414	0.665746	0.541436	0.276243	0.196133	0.0773481	0.0718232	0.0607735
5	0.967136	1.0	0.830986	0.586854	0.356808	0.248826	0.14554	0.0892019
6	0.927461	1.0	0.626943	0.193437	0.0949914	0.0725389	0.0431779	0.0535406
7	0.423611	0.791667	1.0	0.256944	0.0	0.277778	0.465278	0.520833
8	0.716814	0.539823	0.283186	0.129794	0.0648968	0.0766962	0.0265487	0.0324484
9	0.874214	0.849057	0.480084	0.0587002	0.0901468	0.310273	0.387841	0.385744
10	1.0	0.996086	0.694716	0.336595	0.238748	0.268102	0.191781	0.174168
11	0.985507	0.880435	0.518116	0.213768	0.126812	0.119565	0.0833333	0.0797101
12	1.0	0.657563	0.178571	0.178571	0.136555	0.0357143	0.0210084	0.0672269
13	0.986737	0.962865	0.809019	0.35809	0.0822281	0.161804	0.137931	0.0795756
14	1.0	0.912651	0.593373	0.0451807	0.0361446	0.177711	0.316265	0.343374
15	1.0	0.477157	0.0609137	0.0736041	0.0888325	0.0431472	0.035533	0.0279188
16	0.985632	0.78592	0.359195	0.0962644	0.0431034	0.145115	0.196839	0.175287
17	1.0	0.872727	0.358442	0.0571429	0.0649351	0.0909091	0.0649351	0.0493506
18	0.986072	0.506964	0.350975	0.214485	0.133705	0.097493	0.0835655	0.091922
19	1.0	0.93501	0.622642	0.291405	0.236897	0.224319	0.157233	0.140461
20	0.979644	0.974555	0.765903	0.315522	0.0737913	0.160305	0.145038	0.0687023
21	0.950192	0.777778	0.398467	0.0	0.0	0.118774	0.137931	0.229885
22	1.0	0.920245	0.680982	0.466258	0.257669	0.104294	0.0858896	0.110429
23	0.813754	0.719198	0.398281	0.0	0.0974212	0.237822	0.303725	0.340974

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.997118	0.786744	0.0720461	0.092219	0.0835735	0.0893372	0.0691643	0.0835735
25	1.0	0.536341	0.0	0.0626566	0.0626566	0.0651629	0.0451128	0.047619
26	0.993289	0.825503	0.422819	0.154362	0.104027	0.11745	0.0771812	0.0704698
27	0.961694	0.747984	0.135081	0.00201613	0.0725806	0.0725806	0.0625	0.0604839
28	0.989858	0.906694	0.726166	0.553753	0.359026	0.174442	0.0527383	0.0
29	0.975881	0.923933	0.549165	0.230056	0.187384	0.172542	0.116883	0.115028
30	0.962662	0.928571	0.441558	0.0	0.0519481	0.176948	0.181818	0.206169
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [7]:
```
# Show the response variable y

ytrdf
```

Out[7]:
```
87554-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 ⋮
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
```

In [8]:
```
# load the testing data

dfte = CSV.read("mitbih_test.csv", header=false,DataFrame)
```

Out[8]: 21,892 rows × 188 columns (omitted printing of 179 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.758	0.112	0.0	0.0806	0.0785	0.0661	0.0496	0.0475
2	0.908	0.784	0.531	0.363	0.366	0.344	0.333	0.308	0.297
3	0.73	0.212	0.0	0.119	0.102	0.102	0.111	0.124	0.115
4	1.0	0.91	0.681	0.473	0.229	0.0688	0.0	0.00417	0.0146
5	0.57	0.399	0.238	0.148	0.0	0.00336	0.0403	0.0805	0.0705
6	1.0	0.924	0.656	0.196	0.112	0.176	0.122	0.0509	0.0356
7	1.0	0.797	0.321	0.0438	0.0493	0.0658	0.0301	0.00822	0.00548
8	0.909	0.976	0.533	0.134	0.0662	0.0	0.0105	0.0122	0.0314
9	0.928	0.866	0.3	0.0	0.232	0.318	0.275	0.263	0.27
10	1.0	0.914	0.474	0.0	0.0643	0.318	0.405	0.392	0.382
11	0.997	0.861	0.365	0.072	0.0823	0.1	0.0746	0.0514	0.0514
12	1.0	0.934	0.631	0.186	0.127	0.196	0.12	0.0483	0.0534
13	1.0	0.8	0.417	0.116	0.0	0.0931	0.131	0.0931	0.084
14	0.97	1.0	0.523	0.147	0.0606	0.0141	0.00404	0.00606	0.0566
15	1.0	0.91	0.719	0.538	0.326	0.145	0.086	0.113	0.14
16	1.0	0.907	0.526	0.194	0.0746	0.0634	0.0224	0.0299	0.00373
17	1.0	0.747	0.106	0.0709	0.0506	0.0304	0.0101	0.0152	0.0
18	1.0	0.796	0.361	0.0855	0.0723	0.183	0.22	0.189	0.187
19	1.0	0.883	0.191	0.218	0.142	0.128	0.125	0.112	0.125
20	1.0	0.966	0.77	0.573	0.524	0.508	0.467	0.454	0.462
21	0.902	0.902	0.678	0.415	0.224	0.148	0.115	0.109	0.12
22	0.936	0.792	0.513	0.226	0.0404	0.0276	0.0643	0.0864	0.0974
23	0.992	0.876	0.35	0.0	0.125	0.229	0.259	0.296	0.294

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.96	0.755	0.371	0.0906	0.036	0.132	0.178	0.165	0.148
25	1.0	0.892	0.342	0.0	0.151	0.256	0.265	0.283	0.28
26	0.978	0.791	0.0201	0.0221	0.00201	0.0301	0.0803	0.104	0.102
27	0.94	0.994	0.503	0.0508	0.032	0.0904	0.0885	0.0772	0.0791
28	0.792	0.526	0.229	0.161	0.153	0.153	0.132	0.138	0.126
29	1.0	0.904	0.404	0.0339	0.184	0.202	0.146	0.138	0.138
30	0.992	0.785	0.543	0.259	0.0283	0.0	0.0243	0.0243	0.0121
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [9]:
```
# Get all the features for the testing data

Xtedf=dfte[:,1:end-1]
```

Out[9]: 21,892 rows × 187 columns (omitted printing of 178 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.758	0.112	0.0	0.0806	0.0785	0.0661	0.0496	0.0475
2	0.908	0.784	0.531	0.363	0.366	0.344	0.333	0.308	0.297
3	0.73	0.212	0.0	0.119	0.102	0.102	0.111	0.124	0.115
4	1.0	0.91	0.681	0.473	0.229	0.0688	0.0	0.00417	0.0146
5	0.57	0.399	0.238	0.148	0.0	0.00336	0.0403	0.0805	0.0705
6	1.0	0.924	0.656	0.196	0.112	0.176	0.122	0.0509	0.0356
7	1.0	0.797	0.321	0.0438	0.0493	0.0658	0.0301	0.00822	0.00548
8	0.909	0.976	0.533	0.134	0.0662	0.0	0.0105	0.0122	0.0314
9	0.928	0.866	0.3	0.0	0.232	0.318	0.275	0.263	0.27
10	1.0	0.914	0.474	0.0	0.0643	0.318	0.405	0.392	0.382
11	0.997	0.861	0.365	0.072	0.0823	0.1	0.0746	0.0514	0.0514
12	1.0	0.934	0.631	0.186	0.127	0.196	0.12	0.0483	0.0534
13	1.0	0.8	0.417	0.116	0.0	0.0931	0.131	0.0931	0.084
14	0.97	1.0	0.523	0.147	0.0606	0.0141	0.00404	0.00606	0.0566
15	1.0	0.91	0.719	0.538	0.326	0.145	0.086	0.113	0.14
16	1.0	0.907	0.526	0.194	0.0746	0.0634	0.0224	0.0299	0.00373
17	1.0	0.747	0.106	0.0709	0.0506	0.0304	0.0101	0.0152	0.0
18	1.0	0.796	0.361	0.0855	0.0723	0.183	0.22	0.189	0.187
19	1.0	0.883	0.191	0.218	0.142	0.128	0.125	0.112	0.125
20	1.0	0.966	0.77	0.573	0.524	0.508	0.467	0.454	0.462
21	0.902	0.902	0.678	0.415	0.224	0.148	0.115	0.109	0.12
22	0.936	0.792	0.513	0.226	0.0404	0.0276	0.0643	0.0864	0.0974
23	0.992	0.876	0.35	0.0	0.125	0.229	0.259	0.296	0.294

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8	Column9
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.96	0.755	0.371	0.0906	0.036	0.132	0.178	0.165	0.148
25	1.0	0.892	0.342	0.0	0.151	0.256	0.265	0.283	0.28
26	0.978	0.791	0.0201	0.0221	0.00201	0.0301	0.0803	0.104	0.102
27	0.94	0.994	0.503	0.0508	0.032	0.0904	0.0885	0.0772	0.0791
28	0.792	0.526	0.229	0.161	0.153	0.153	0.132	0.138	0.126
29	1.0	0.904	0.404	0.0339	0.184	0.202	0.146	0.138	0.138
30	0.992	0.785	0.543	0.259	0.0283	0.0	0.0243	0.0243	0.0121
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [10]:
```
# Get the response variable y of the testing data

ytedf=dfte[:,end]
```

Out[10]:
```
21892-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 ⋮
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
```

In [11]:
```
# Convert "Dataframe" to "Matrix" as the data with Dataframe datatype can not be plot out

Xtrm = Matrix(Xtrdf)
```

Out[11]:

```
87554×187 Matrix{Float64}:
 0.977941  0.926471  0.681373  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.960114  0.863248  0.461538     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0       0.659459  0.186486     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.925414  0.665746  0.541436     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.967136  1.0       0.830986     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.927461  1.0       0.626943  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.423611  0.791667  1.0          0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.716814  0.539823  0.283186     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.874214  0.849057  0.480084     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0       0.996086  0.694716     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.985507  0.880435  0.518116  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0       0.657563  0.178571     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.986737  0.962865  0.809019     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 ⋮                              ⋱  ⋮                       ⋮
 0.654854  0.567026  0.454545     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.76112   0.662273  0.537068     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.621951  0.521341  0.463415     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.799242  0.515152  0.545455  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.757235  0.663987  0.561093     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.717325  0.62766   0.534954     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 1.0       0.405594  0.440559     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.807018  0.494737  0.536842     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.718333  0.605     0.486667  …  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.906122  0.62449   0.595918     0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.858228  0.64557   0.84557      0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.901506  0.845886  0.800695     0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

In [12]:

```
# In the training dataset, create 5 sub-datasets based on the 5 different heartbeat types

# The Heartbeat types include 5 classes which are N: normal, S: supraventricular, V: ventricular, F: fusion, and Q: unk
# They are represented by the numbers 0.0,1.0,2.0,3.0,4.0 separately

# For example,"itr0" means in the training dataset, the index of the heartbeat type which is the N:"normal" type.

itr0=findall(ytrdf -> ytrdf == 0.0,ytrdf)
itr1=findall(ytrdf -> ytrdf == 1.0,ytrdf)
itr2=findall(ytrdf -> ytrdf == 2.0,ytrdf)
itr3=findall(ytrdf -> ytrdf == 3.0,ytrdf)
itr4=findall(ytrdf -> ytrdf == 4.0,ytrdf)

size(itr0)[1]+size(itr1)[1]+size(itr2)[1]+size(itr3)[1]+size(itr4)[1]==size(ytrdf)[1]

# To make sure the total size of the subsets equal to the total size of the training data
```

Out[12]: true

In [13]:
```
# Response variable y for different heartbeat types in the training dataset

ytr0=ytrdf[itr0,:]
ytr1=ytrdf[itr1,:]
ytr2=ytrdf[itr2,:]
ytr3=ytrdf[itr3,:]
ytr4=ytrdf[itr4,:]

size(ytr0)[1]+size(ytr1)[1]+size(ytr2)[1]+size(ytr3)[1]+size(ytr4)[1]==size(ytrdf)[1]
```

Out[13]: true

In [14]:
```
#The features X for different heartbeat types in the training dataset

Xtr0=Xtrm[itr0,:]
Xtr1=Xtrm[itr1,:]
Xtr2=Xtrm[itr2,:]
Xtr3=Xtrm[itr3,:]
Xtr4=Xtrm[itr4,:]

size(Xtr0)[1]+size(Xtr1)[1]+size(Xtr2)[1]+size(Xtr3)[1]+size(Xtr4)[1]==size(Xtrm)[1]
```

Out[14]: true

In [ ]:
```
```

In [15]:
```
# Part 2: Exploratory data analysis
```

In [16]:
```
#1.Carry out statistical comparison of the training dataset and testing dataset

using Statistics
```

In [17]:
```
# Compare the mean and the standard deviation on the training dataset and testing dataset
```

In [18]:
```
mean(ytrdf),std(ytrdf)
```

Out[18]: (0.4733764305457204, 1.1431844061643537)

In [19]:
```
mean(ytedf),std(ytedf)
```

Out[19]:
```
(0.47368901881966013, 1.1434469260642504)
```

In [ ]:
```
# By comparison, the corresponding mean and std values of the training dataset and testing dataset are found to be simi
```

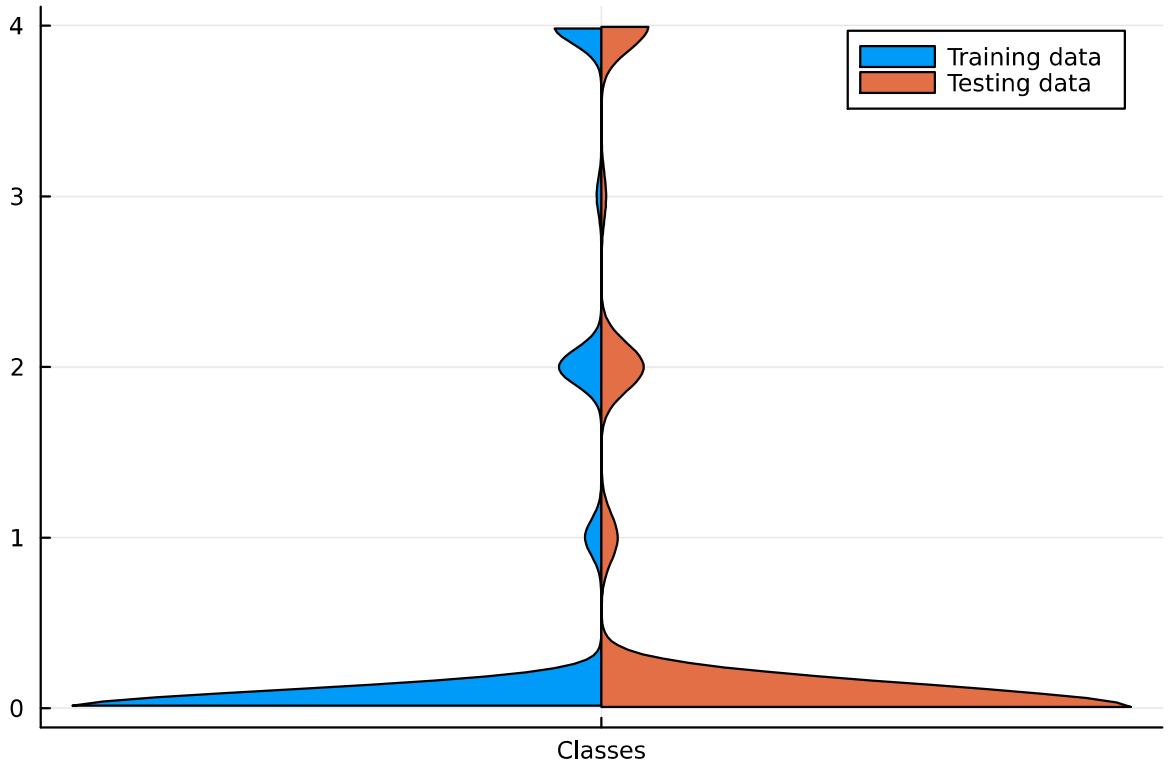
In [20]:
```
#2.Plot the training dataset and the testing dataset

using StatsPlots

violin(["Classes"], ytrdf, side=:left, label="Training data")
violin!(["Classes"], ytedf, side=:right, label="Testing data")

# We can see that the data density distribution of the five different heartbeat types is similar in the training datase
```

Out[20]:



In [21]:
```
# 3.Perform statistical tests on the response variable y in the training dataset and the testing dataset
```

In [22]:
```
import Pkg; Pkg.add("HypothesisTests")

using HypothesisTests
```

```
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
```

In [23]:
```
UnequalVarianceTTest(ytrdf, ytedf)
```

Out[23]:
```
Two sample t-test (unequal variance)
------------------------------------
Population details:
    parameter of interest:   Mean difference
    value under h_0:         0
    point estimate:          -0.000312588
    95% confidence interval: (-0.01725, 0.01662)

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.9711

Details:
    number of observations:   [87554,21892]
    t-statistic:              -0.036179068656426235
    degrees of freedom:       33674.67479087656
    empirical standard error: 0.008640030977807921
```

In [24]:
```
MannWhitneyUTest(ytrdf, ytedf)
```

Out[24]:
```
Approximate Mann-Whitney U test
-------------------------------
Population details:
    parameter of interest:   Location parameter (pseudomedian)
    value under h_0:         0
    point estimate:          0.0

Test summary:
    outcome with 95% confidence: fail to reject h_0
    two-sided p-value:           0.9668

Details:
    number of observations in each group: [87554, 21892]
    Mann-Whitney-U statistic:             9.58252e8
    rank sums:                            [4.17228e9, 1.81699e9]
    adjustment for ties:                  7.44327e14
    normal approximation (μ, σ):          (-114312.0, 2.74887e6)
```

In [25]:
```
# Compare the test results, we can see that both tests return the same "fail to reject h_0" outcome.

# In short, the p-value (greater than 0.05) indicates that there is at least 95% confidence that the training dataset a
```

In [ ]:
```

```

In [26]:
```
# 4.Get and visual a sample heartbeat type in each of the 5 heartbeat types.

using StatsBase

countmap(ytrdf)
```

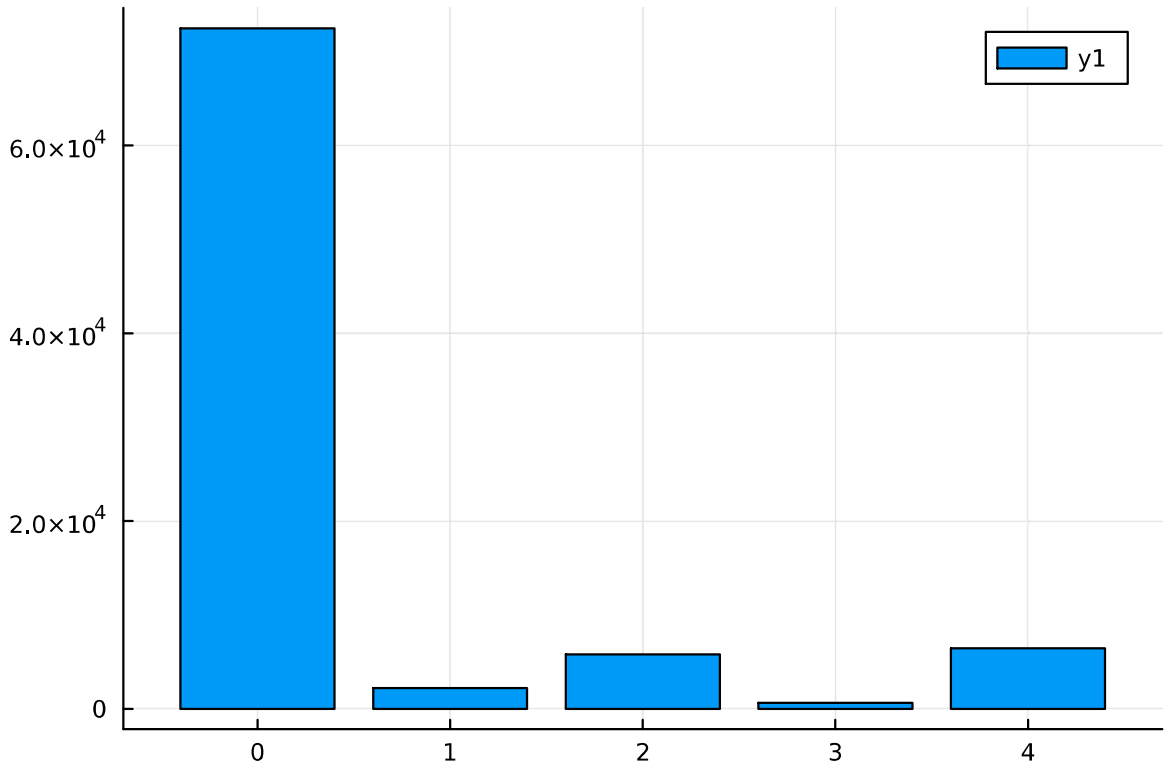
Out[26]:
```
Dict{Float64, Int64} with 5 entries:
  0.0 => 72471
  4.0 => 6431
  2.0 => 5788
  3.0 => 641
  1.0 => 2223
```

In [27]:
```
bar(countmap(ytrdf))

#From the bar charts, we can find that the normal heartbeat type 0 is a lot more than the other 4 problematic heartbeat

#And the number of ventricular ectopic heartbeat type 2 and unknown heartbeat type 4 is similar
```

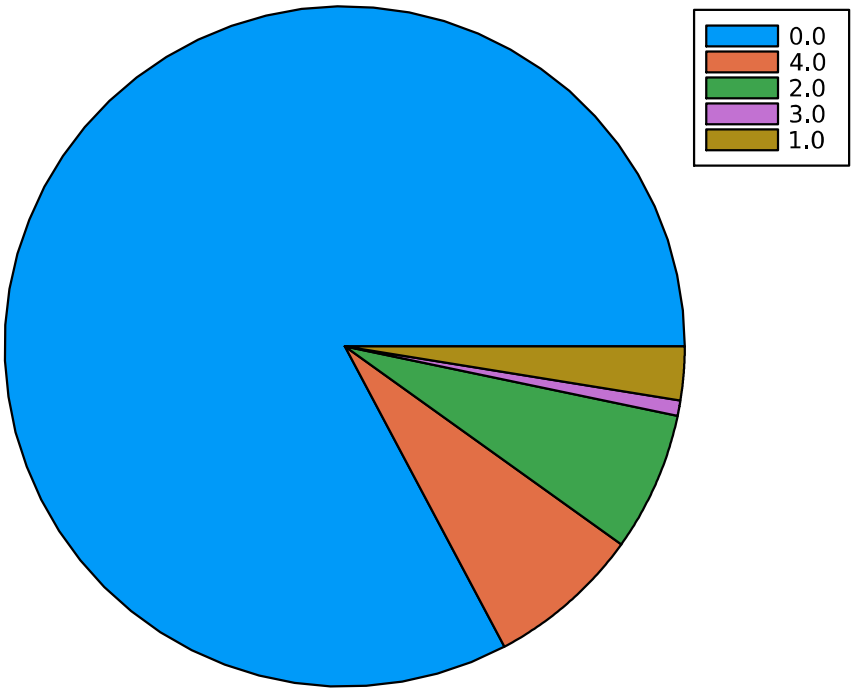
Out[27]:



In [28]:
```
pie(countmap(ytrdf))

# From the pie chart, we can see that the data under different heartbeat types are highly imbalance, need to make the d
```

Out[28]:



In [29]:

```
#The percentage of the total number of each heartbeat type in the training dataset

println("The heartbeat type N is ", join([round(length(ytr0)/length(ytrdf); digits=4)*100,"%"]))
println("The heartbeat type S is ", join([round(length(ytr1)/length(ytrdf); digits=4)*100,"%"]))
println("The heartbeat type V is ", join([round(length(ytr2)/length(ytrdf); digits=4)*100,"%"]))
println("The heartbeat type F is ", join([round(length(ytr3)/length(ytrdf); digits=4)*100,"%"]))
println("The heartbeat type Q is ", join([round(length(ytr4)/length(ytrdf); digits=4)*100,"%"]))

# Since the data in different heartbeat type is highly imbalanced, we need to make the training dataset balance before
```

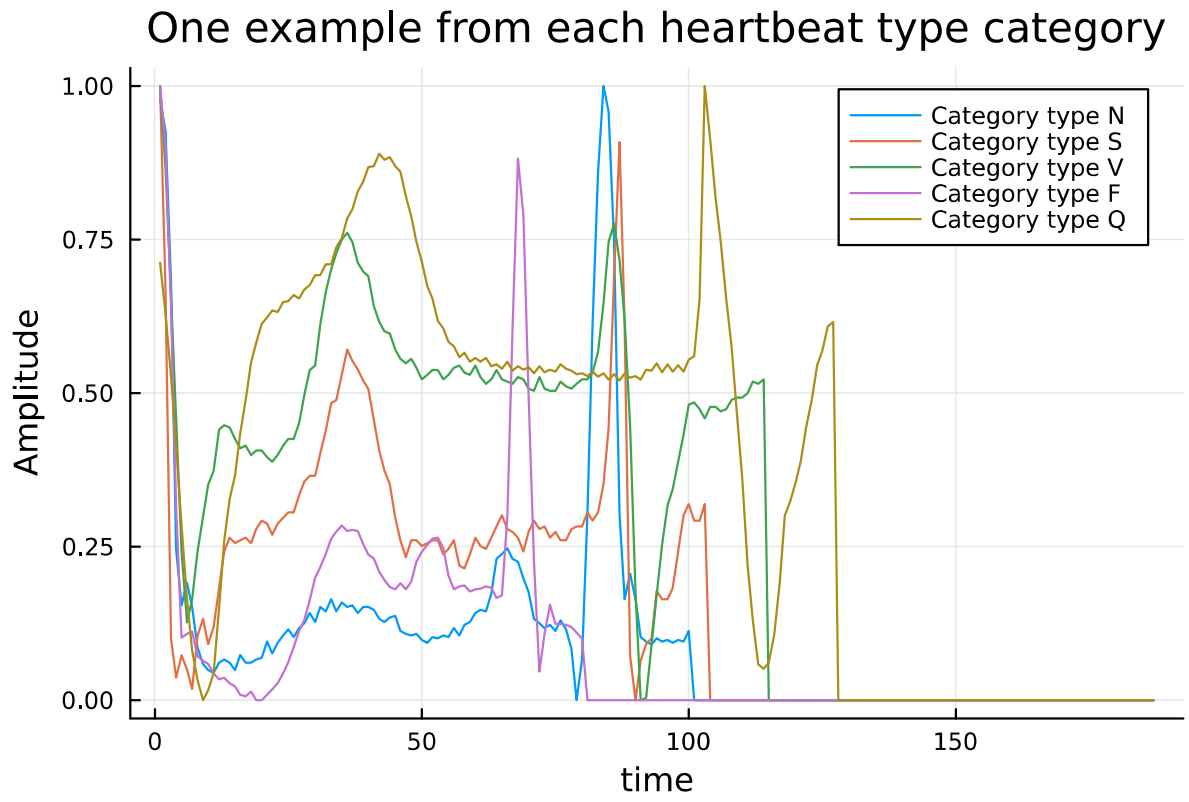
```
The heartbeat type N is 82.77%
The heartbeat type S is 2.54%
The heartbeat type V is 6.61%
The heartbeat type F is 0.73%
The heartbeat type Q is 7.35%
```

In [ ]:

In [30]:
```
# 5.Visualize a sample of the different heartbeat types in one plotting

plot(Xtr0[1,:], xlabel="time",ylabel="Amplitude",title="One example from each heartbeat type category",label="Category
plot!(Xtr1[1,:],xlabel="time",ylabel="Amplitude",label="Category type S")
plot!(Xtr2[2,:],xlabel="time",ylabel="Amplitude",label="Category type V")
plot!(Xtr3[1,:],xlabel="time",ylabel="Amplitude",label="Category type F")
plot!(Xtr4[1,:],xlabel="time",ylabel="Amplitude",label="Category type Q")
```

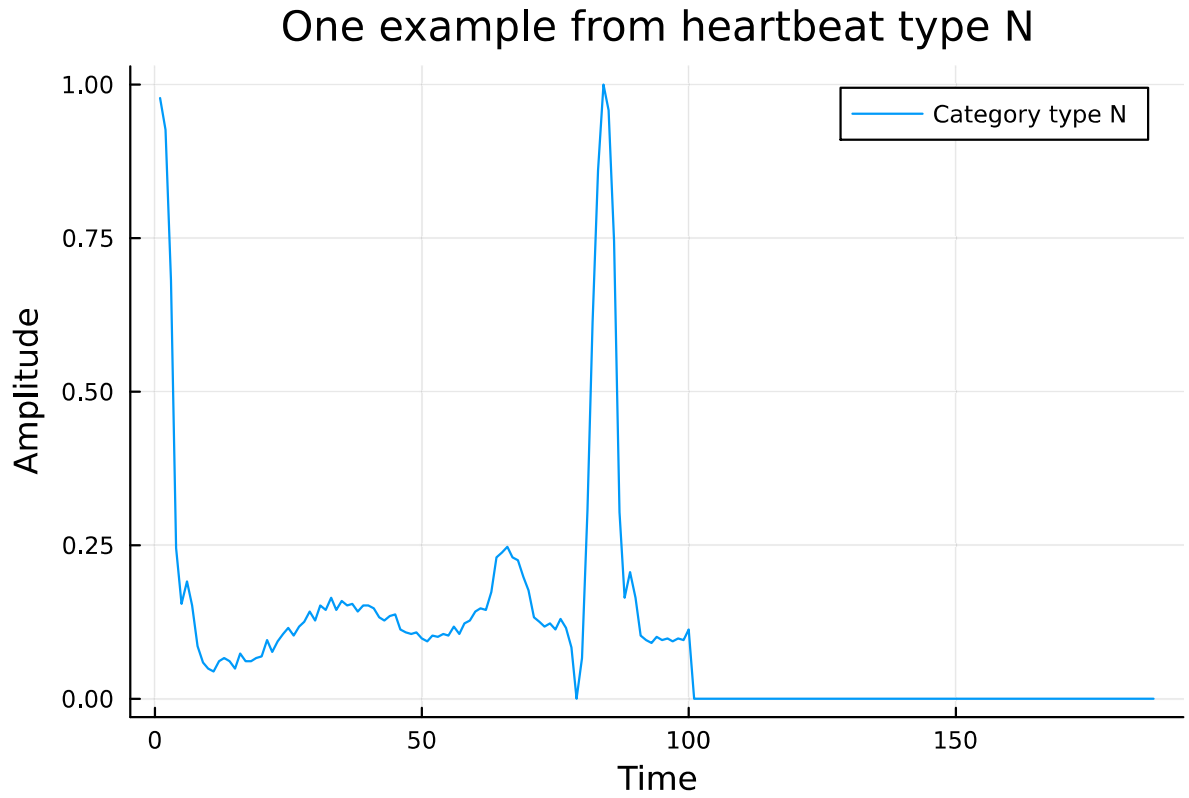
Out[30]:



In [169…
```
# Plot one example of the normal heartbeat type

plot(Xtr0[1,:], xlabel="Time",ylabel="Amplitude",title="One example from heartbeat type N",label="Category type N")
```

Out[169]:



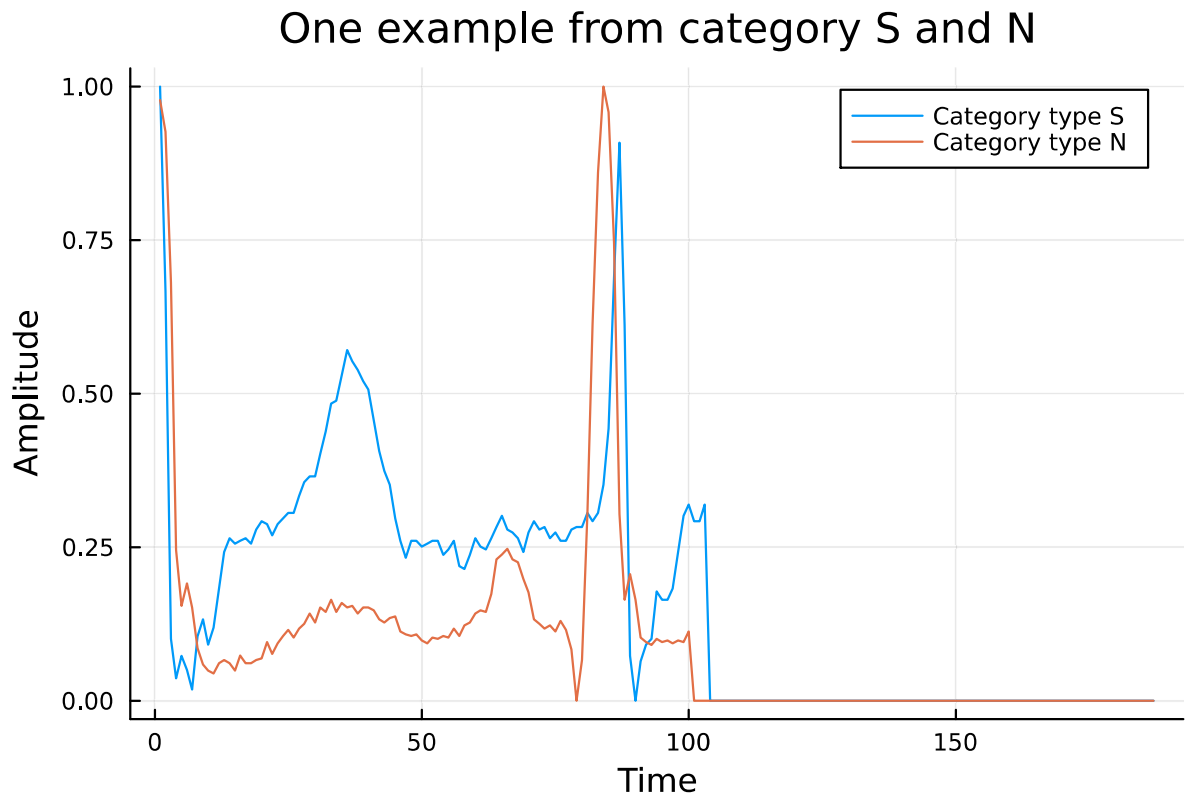
In [32]:

```
# Compare the Plots of supra ventricular ectopic heartbeat type with the normal heartbeat type

plot(Xtr1[1,:],xlabel="Time",ylabel="Amplitude",title="One example from category S and N",label="Category type S")
plot!(Xtr0[1,:], xlabel="Time",ylabel="Amplitude",label="Category type N")

# We can see the shape of the curve is different, after compare with these 2 different heartbeat types.
# In the supra ventricular ectopic heartbeat type,the t wave amplitude is higher, the t wave duration is similar to the
# The amplitude of p wave is small, there is no obvious p ware, the timing of R waves are similar to the normal heartbe
# In the supra ventricular ectopic heartbeat type,the RR interval is similar to the normal heartbeat type's RR interval
```

Out[32]:



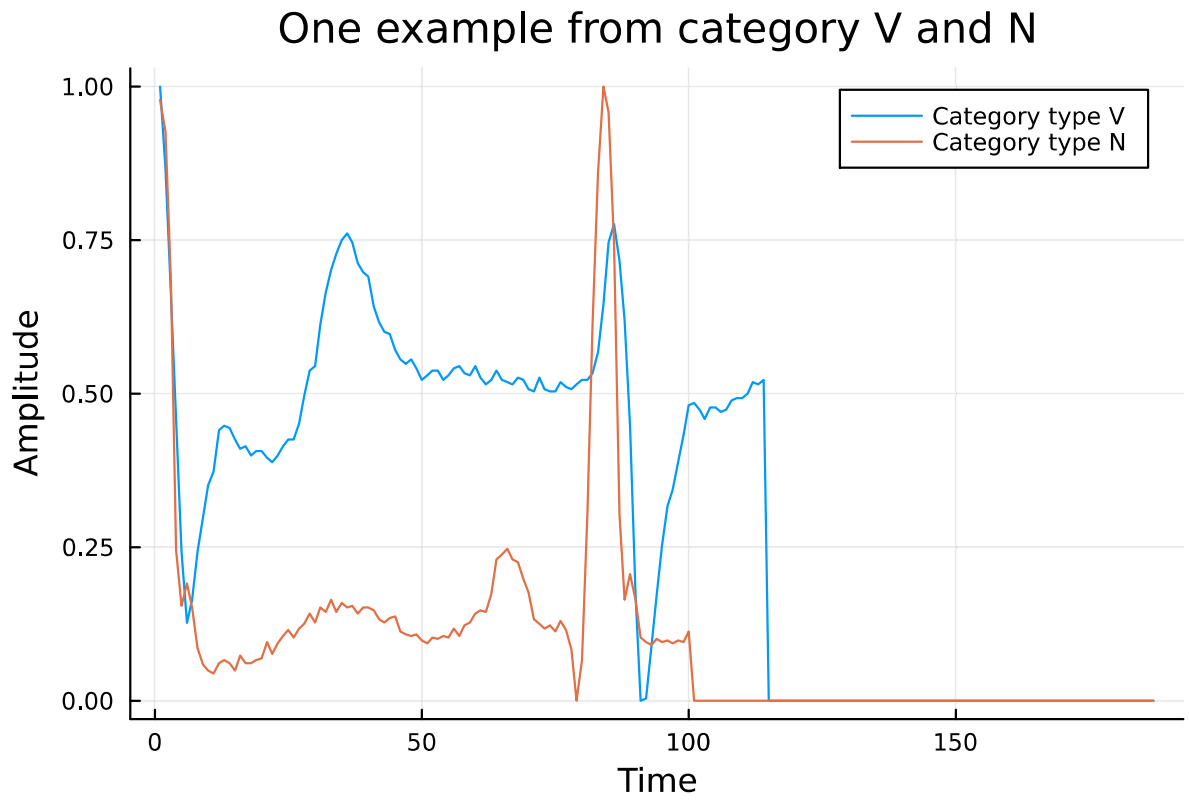
In [33]:

```
# Plot of ventricular ectopic heartbeat type compare with the normal heartbeat type

plot(Xtr2[2,:],xlabel="Time",ylabel="Amplitude",title="One example from category V and N",label="Category type V")
plot!(Xtr0[1,:], xlabel="Time",ylabel="Amplitude",label="Category type N")

# We can see the shape of the curve is different,after compare with these 2 different heartbeat types.
# In the ventricular ectopic heartbeat type plot, the t wave amplitude is higher, the t wave duration is shorter compar
# In ventricular ectopic heartbeat type,the p wave starts earlier and  looks higher
# The r wave timing is similar under the 2 different heartbeat types, but the amplitude in ventricular ectopic heartbea
```

Out[33]:



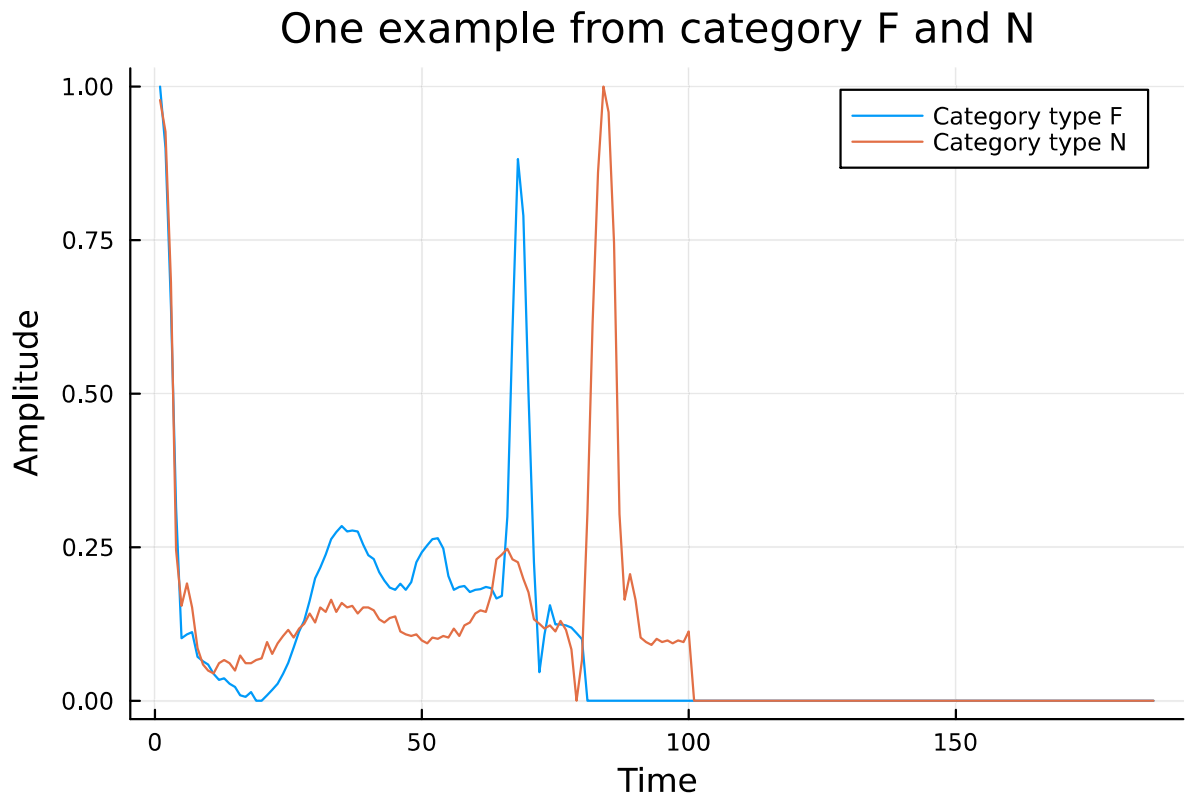
In [34]:

```
# Plot of fusion heartbeat type compare with the normal heartbeat type

plot(Xtr3[1,:],xlabel="Time",ylabel="Amplitude",title="One example from category F and N",label="Category type F")
plot!(Xtr0[1,:], xlabel="Time",ylabel="Amplitude",label="Category type N")

# We can see the shape of the curve is different,after compare with these 2 different heartbeat types.
# The t wave timing is later and the duration shorter in the fusion heartbeat type
# The timing of p wave is earlier, and the timing of R wave is earlier in the fusion heartbeat type
# The RR interval n the fusion heartbeat type is shorter than the normal heart beat's RR interval
```

Out[34]:



In [35]:

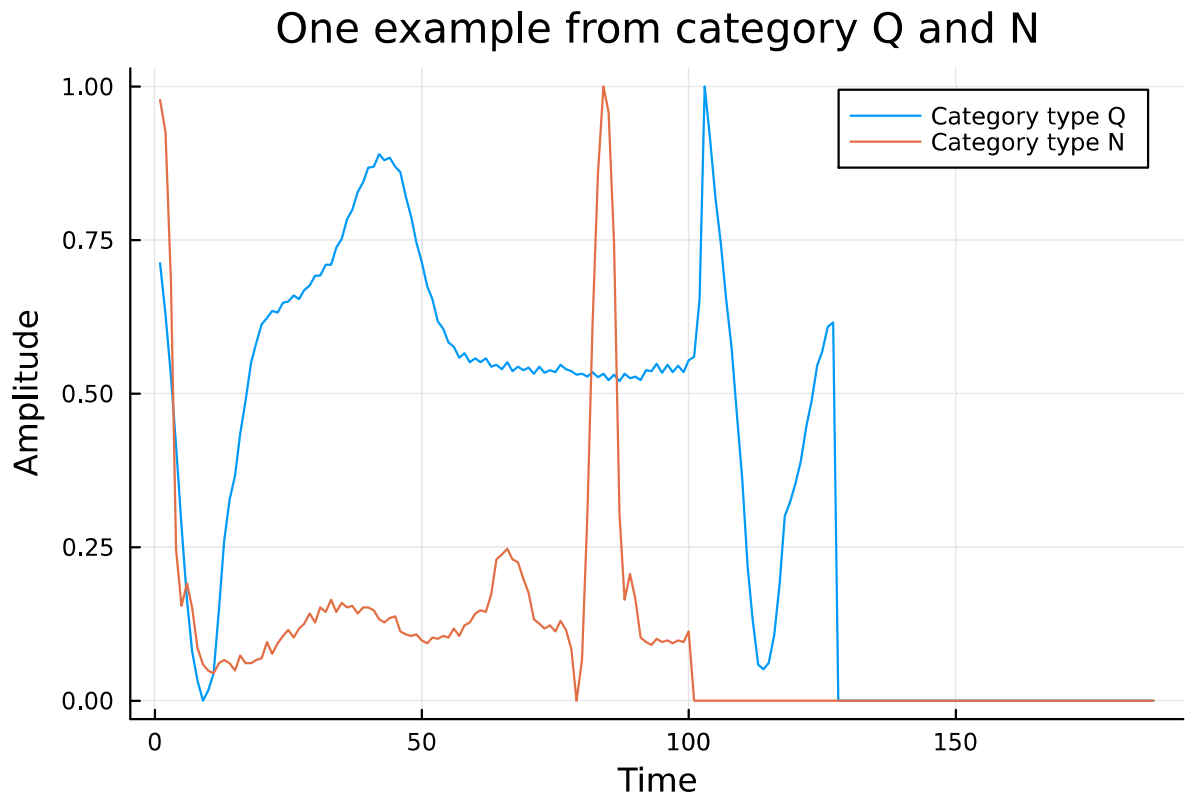
```
# Plot of unknown heartbeat type compare with the normal heartbeat type

plot(Xtr4[1,:],xlabel="Time",ylabel="Amplitude",title="One example from category Q and N",label="Category type Q")
plot!(Xtr0[1,:], xlabel="Time",ylabel="Amplitude",label="Category type N")

# We can see the shape of the curve is different,after compare with these 2 different heartbeat types.

# Any type that is not classifed as the 4 differnt heartbeat types (N,F,V,S), are classfied as this class "unknown"
```

Out[35]:



In [36]:
```
# From the above comparision plots, we can see that the features which we can extract, should relate to the length, tim
```

In [ ]:
```
```

In [37]:
```
# 6.Clustering Analysis
```

In [38]:
```
import Pkg; Pkg.add("StatsBase")
using Statistics, StatsBase, LinearAlgebra

import Pkg; Pkg.add("Distances")
using Distances

import Pkg; Pkg.add("Clustering")
using Clustering

using Statistics
```

```
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
```

In [39]:
```
# Kmeans clustering

km=kmeans(Xtrm', 5; maxiter=100)

# Group all data into 5 clusters as there is 5 different heartbeat types.
```

Out[39]:
```
KmeansResult{Matrix{Float64}, Float64, Int64}([0.3235231123942794 0.9693992332339523 … 0.958473662564486 0.911071461969
8351; 0.3550917502527527 0.7686204641923863 … 0.833886535156989 0.7841846037331194; … ; 0.0046817521451984565 0.0004583
4884168267985 … 0.0007436506652448965 0.0; 0.004414183535616645 0.0004384113946469748 … 0.0006455993918730067 0.0], [4,
2, 2, 2, 2, 3, 1, 2, 3, 2  …  3, 4, 1, 1, 2, 4, 1, 3, 4, 1], [2.4290303785134046, 2.95179746776323, 1.9830228134612735,
2.76535394929176, 4.650591729976448, 3.3098872497659784, 3.2830078976959953, 3.9857285760798504, 2.8651985649194742, 2.
7648928821911714  …  2.8976966570930998, 2.8728019705450016, 4.6554752084539786, 5.016026730220844, 3.4897176653950384,
2.233253192308325, 5.919047523409844, 5.163309068216485, 3.3630897510477755, 6.119567796869944], [7133, 28967, 17843, 1
8454, 15157], [7133, 28967, 17843, 18454, 15157], 305300.8220353245, 54, true)
```

In [40]:
```
a = assignments(km)

# It can be used as a feature for predicting classification
```

Out[40]:
```
87554-element Vector{Int64}:
 4
 2
 2
 2
 2
 3
 1
 2
 3
 2
 2
 3
 2
 ⋮
 3
 1
 3
 4
 1
 1
 2
 4
 1
 3
 4
 1
```

In [41]:
```
countmap(a)

# Find that the training classification proportion seems not the same as the clustering proportion
```

Out[41]:
```
Dict{Int64, Int64} with 5 entries:
  5 => 15157
  4 => 18454
  2 => 28967
  3 => 17843
  1 => 7133
```

In [ ]:
```
# Cluster the training data into 5 groups as there is 5 heartbeat types

# Get the numbers of samples that belong to each kmean group

# Compare with the number of samples in each heartbeat type
```

```
# Noticed the numbers are different

# This means the classification of the ECG data does not totally depends on the distance between samples

# But worth to use kmeans cluster number as a feature later on during the feature extraction process
```

In [ ]:

In [42]:
```
# Part 3: Heartbeat type classification

import Pkg; Pkg.add("MLJ")
using MLJ
```

```
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
```

In [43]:
```
# In general, we can use either AUC or Accuracy on the training data to tune the model, find the best hyperparameters,

# AUC is good at handling imbalanced data, but AUC is only for binary classification. Since we have 5 heartbeat type cl

# Therefore we need to use Accuracy to evaluate the performance on training data and testing data.

# In order for us to use Accuracy, the prerequisite is the training dataset needs to be balanced.

# If we use an imbalanced dataset, the classification accuracy is not reliable.

# For example, if the training dataset is imbalanced with 80% normal type data, a naive classifier which built from the
```

In [44]:
```
# Make the training dataset balance
```

In [45]:
```
import Pkg; Pkg.add("MLUtils")
using MLUtils
```

```
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
```

In [46]:
```
MLUtils.getobs(dftr::DataFrame, i) = dftr[i,:]
MLUtils.numobs(dftr::DataFrame) = nrow(dftr)
```

In [47]:
```
bdftr=getobs(undersample(dftr, dftr.Column188))
```

Out[47]: 3,205 rows × 188 columns (omitted printing of 180 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.0789474	0.0845865
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.11315	0.11315
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.281457	0.284768
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.51505	0.438127
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.709641	0.470852
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.0957854	0.103448
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.454545	0.422925
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.40315	0.401575
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.132159	0.0969163
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.0327613	0.0
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.092511	0.061674
12	0.962963	1.0	0.83878	0.337691	0.0718954	0.246187	0.400871	0.320261
13	0.990826	1.0	0.454128	0.0917431	0.0688073	0.12844	0.0963303	0.0825688
14	0.9631	0.512915	0.568266	0.557196	0.575646	0.546125	0.546125	0.476015
15	1.0	0.947566	0.273408	0.0	0.101124	0.172285	0.11985	0.168539
16	0.986708	0.93865	0.875256	0.803681	0.687117	0.537832	0.380368	0.276074
17	0.915865	0.879808	0.769231	0.610577	0.454327	0.399038	0.372596	0.401442
18	0.929648	0.68593	0.211055	0.0125628	0.00502513	0.0351759	0.0251256	0.00753769
19	0.527964	0.346756	0.313199	0.304251	0.261745	0.239374	0.212528	0.174497
20	0.940994	0.680124	0.13354	0.0869565	0.15528	0.124224	0.0993789	0.0962733
21	0.858456	0.880515	0.931985	1.0	0.985294	0.639706	0.349265	0.235294
22	0.948598	0.831776	0.200935	0.00934579	0.0420561	0.0934579	0.116822	0.130841
23	0.945183	0.877076	0.252492	0.0	0.232558	0.318937	0.277409	0.257475

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.906306	0.836036	0.563964	0.174775	0.0	0.0522523	0.113514	0.158559
25	0.90566	0.733491	0.141509	0.0542453	0.0518868	0.00707547	0.0188679	0.00471698
26	1.0	0.983687	0.807504	0.525285	0.336052	0.24633	0.17292	0.153344
27	0.975664	0.809735	0.143805	0.123894	0.265487	0.265487	0.223451	0.238938
28	1.0	0.93913	0.898551	0.75942	0.652174	0.602899	0.536232	0.513043
29	0.620112	0.586592	0.416201	0.150838	0.0	0.0586592	0.153631	0.192737
30	0.901449	0.692754	0.542029	0.321739	0.191304	0.115942	0.101449	0.0637681
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [48]:
```
bdftr
```

Out[48]: 3,205 rows × 188 columns (omitted printing of 180 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.0789474	0.0845865
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.11315	0.11315
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.281457	0.284768
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.51505	0.438127
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.709641	0.470852
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.0957854	0.103448
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.454545	0.422925
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.40315	0.401575
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.132159	0.0969163
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.0327613	0.0
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.092511	0.061674
12	0.962963	1.0	0.83878	0.337691	0.0718954	0.246187	0.400871	0.320261
13	0.990826	1.0	0.454128	0.0917431	0.0688073	0.12844	0.0963303	0.0825688
14	0.9631	0.512915	0.568266	0.557196	0.575646	0.546125	0.546125	0.476015
15	1.0	0.947566	0.273408	0.0	0.101124	0.172285	0.11985	0.168539
16	0.986708	0.93865	0.875256	0.803681	0.687117	0.537832	0.380368	0.276074
17	0.915865	0.879808	0.769231	0.610577	0.454327	0.399038	0.372596	0.401442
18	0.929648	0.68593	0.211055	0.0125628	0.00502513	0.0351759	0.0251256	0.00753769
19	0.527964	0.346756	0.313199	0.304251	0.261745	0.239374	0.212528	0.174497
20	0.940994	0.680124	0.13354	0.0869565	0.15528	0.124224	0.0993789	0.0962733
21	0.858456	0.880515	0.931985	1.0	0.985294	0.639706	0.349265	0.235294
22	0.948598	0.831776	0.200935	0.00934579	0.0420561	0.0934579	0.116822	0.130841
23	0.945183	0.877076	0.252492	0.0	0.232558	0.318937	0.277409	0.257475

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.906306	0.836036	0.563964	0.174775	0.0	0.0522523	0.113514	0.158559
25	0.90566	0.733491	0.141509	0.0542453	0.0518868	0.00707547	0.0188679	0.00471698
26	1.0	0.983687	0.807504	0.525285	0.336052	0.24633	0.17292	0.153344
27	0.975664	0.809735	0.143805	0.123894	0.265487	0.265487	0.223451	0.238938
28	1.0	0.93913	0.898551	0.75942	0.652174	0.602899	0.536232	0.513043
29	0.620112	0.586592	0.416201	0.150838	0.0	0.0586592	0.153631	0.192737
30	0.901449	0.692754	0.542029	0.321739	0.191304	0.115942	0.101449	0.0637681
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [49]:
```
bytrdf, bXtrdf = unpack(bdftr, ==(:Column188),name->true)
```

Out[49]: ([3.0, 2.0, 2.0, 4.0, 2.0, 1.0, 4.0, 2.0, 4.0, 4.0 … 1.0, 4.0, 3.0, 2.0, 1.0, 1.0, 3.0, 1.0, 2.0, 2.0], **3205×187 DataFrame**

Row	Column1	Column2	Column3	Column4	Column5	Column6	Column ⋯
	Float64	Float64	Float64	Float64	Float64	Float64	Float6 ⋯
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.0789 ⋯
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.1131
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.2814
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.5150
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.7096 ⋯
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.0957
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.4545
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.4031
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.1321 ⋯
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.0327
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.0925
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮ ⋱
3196	1.0	0.937743	0.315175	0.0	0.0700389	0.167315	0.1673
3197	1.0	0.537815	0.554622	0.588235	0.60084	0.592437	0.5840 ⋯
3198	1.0	0.695238	0.235714	0.0452381	0.0166667	0.0238095	0.0047
3199	0.852804	0.82243	0.752336	0.607477	0.462617	0.373832	0.3504
3200	0.967643	0.964561	0.412943	0.0	0.281972	0.359014	0.3343
3201	0.979557	0.938671	0.417376	0.105622	0.0459966	0.0	0.0085 ⋯
3202	0.386364	0.393939	0.200758	0.162879	0.0757576	0.0	0.0492
3203	0.97992	0.891566	0.261044	0.0	0.0843373	0.168675	0.1325
3204	0.0	0.0833333	0.173333	0.313333	0.416667	0.413333	0.43
3205	0.923664	0.882952	0.821883	0.798982	0.760814	0.605598	0.3944 ⋯

181 columns and 3184 rows omitted, **0×0 DataFrame**)

In [50]:
```
countmap(bytrdf)

# The dataset is balance now, it can be used for model building
```

Out[50]:
```
Dict{Float64, Int64} with 5 entries:
  0.0 => 641
  4.0 => 641
  2.0 => 641
  3.0 => 641
  1.0 => 641
```

In [51]:
```
# Show the new balanced training data with all the features

bXtrdf
```

Out[51]: 3,205 rows × 187 columns (omitted printing of 179 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.0789474	0.0845865
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.11315	0.11315
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.281457	0.284768
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.51505	0.438127
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.709641	0.470852
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.0957854	0.103448
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.454545	0.422925
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.40315	0.401575
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.132159	0.0969163
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.0327613	0.0
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.092511	0.061674
12	0.962963	1.0	0.83878	0.337691	0.0718954	0.246187	0.400871	0.320261
13	0.990826	1.0	0.454128	0.0917431	0.0688073	0.12844	0.0963303	0.0825688
14	0.9631	0.512915	0.568266	0.557196	0.575646	0.546125	0.546125	0.476015
15	1.0	0.947566	0.273408	0.0	0.101124	0.172285	0.11985	0.168539
16	0.986708	0.93865	0.875256	0.803681	0.687117	0.537832	0.380368	0.276074
17	0.915865	0.879808	0.769231	0.610577	0.454327	0.399038	0.372596	0.401442
18	0.929648	0.68593	0.211055	0.0125628	0.00502513	0.0351759	0.0251256	0.00753769
19	0.527964	0.346756	0.313199	0.304251	0.261745	0.239374	0.212528	0.174497
20	0.940994	0.680124	0.13354	0.0869565	0.15528	0.124224	0.0993789	0.0962733
21	0.858456	0.880515	0.931985	1.0	0.985294	0.639706	0.349265	0.235294
22	0.948598	0.831776	0.200935	0.00934579	0.0420561	0.0934579	0.116822	0.130841
23	0.945183	0.877076	0.252492	0.0	0.232558	0.318937	0.277409	0.257475

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.906306	0.836036	0.563964	0.174775	0.0	0.0522523	0.113514	0.158559
25	0.90566	0.733491	0.141509	0.0542453	0.0518868	0.00707547	0.0188679	0.00471698
26	1.0	0.983687	0.807504	0.525285	0.336052	0.24633	0.17292	0.153344
27	0.975664	0.809735	0.143805	0.123894	0.265487	0.265487	0.223451	0.238938
28	1.0	0.93913	0.898551	0.75942	0.652174	0.602899	0.536232	0.513043
29	0.620112	0.586592	0.416201	0.150838	0.0	0.0586592	0.153631	0.192737
30	0.901449	0.692754	0.542029	0.321739	0.191304	0.115942	0.101449	0.0637681
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [52]:
```
# Show the new balanced training data with all the response variable y

bytrdf
```

Out[52]:
```
3205-element Vector{Float64}:
 3.0
 2.0
 2.0
 4.0
 2.0
 1.0
 4.0
 2.0
 4.0
 4.0
 3.0
 2.0
 0.0
 ⋮
 0.0
 3.0
 1.0
 4.0
 3.0
 2.0
 1.0
 1.0
 3.0
 1.0
 2.0
 2.0
```

In [53]:
```
# Make the balanced training data response variable y to be categority for later machine learning classification use

# If do not convert to CategoricalArrays, it will have error message later on

using CategoricalArrays

bytrdf = CategoricalArray(bytrdf)
```

Out[53]:
```
3205-element CategoricalArray{Float64,1,UInt32}:
 3.0
 2.0
 2.0
 4.0
 2.0
 1.0
 4.0
 2.0
 4.0
 4.0
 3.0
 2.0
 0.0
 ⋮
 0.0
 3.0
 1.0
 4.0
 3.0
 2.0
 1.0
 1.0
 3.0
 1.0
 2.0
 2.0
```

In [54]:
```
# Machine learning classification algorithm 1. kNN -- no need feature extraction, quick
```

In [55]:
```
using NearestNeighborModels

import Pkg; Pkg.add("MLJModels")
import Pkg; Pkg.add("CategoricalArrays")
import Pkg; Pkg.add("MLJBase")

using MLJModels, CategoricalArrays, MLJBase
```

```
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
  Resolving package versions...
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
 No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
```

In [56]:

```
# Performance tuning: use 10 folds cross validation to build models, using k = 1 to 10

# To tune the model and select the best hyperparameters that can create the model which has the highest accuracy

using Random

Random.seed!(12)

cv=CV(nfolds=10)

perftrknn1 = evaluate!(machine(KNNClassifier(K=1), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn2 = evaluate!(machine(KNNClassifier(K=2), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn3 = evaluate!(machine(KNNClassifier(K=3), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn4 = evaluate!(machine(KNNClassifier(K=4), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn5 = evaluate!(machine(KNNClassifier(K=5), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn6 = evaluate!(machine(KNNClassifier(K=6), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn7 = evaluate!(machine(KNNClassifier(K=7), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn8 = evaluate!(machine(KNNClassifier(K=8), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn9 = evaluate!(machine(KNNClassifier(K=9), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrknn10 = evaluate!(machine(KNNClassifier(K=10), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
```

Out[56]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict_mode	0.83	0.0115	[0.85, 0.81, 0.813, 0.83 ⋯

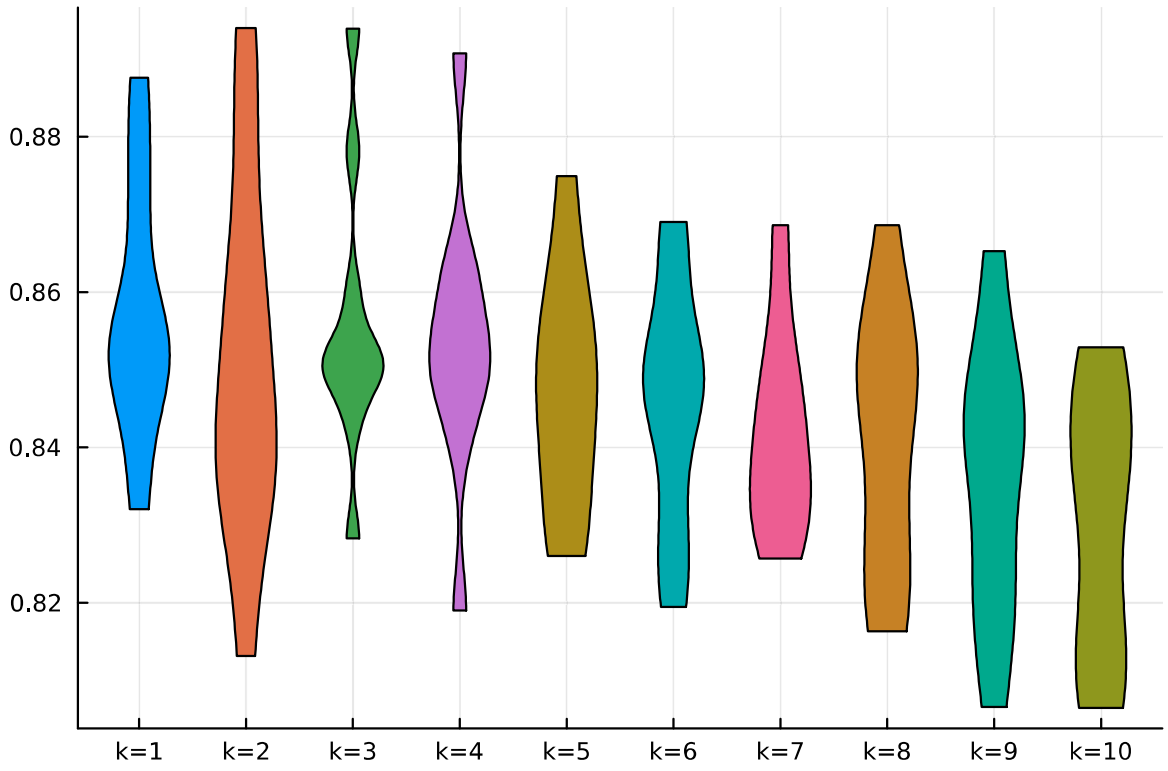
1 column omitted

In [170…

```
# Plot the 10 folds cross validation accuracy
```

```
violin(["k=1"], perftrknn1.per_fold, label=nothing)
violin!(["k=2"], perftrknn2.per_fold, label=nothing)
violin!(["k=3"], perftrknn3.per_fold, label=nothing)
violin!(["k=4"], perftrknn4.per_fold, label=nothing)
violin!(["k=5"], perftrknn5.per_fold, label=nothing)
violin!(["k=6"], perftrknn6.per_fold, label=nothing)
violin!(["k=7"], perftrknn7.per_fold, label=nothing)
violin!(["k=8"], perftrknn8.per_fold, label=nothing)
violin!(["k=9"], perftrknn9.per_fold, label=nothing)
violin!(["k=10"], perftrknn10.per_fold, label=nothing)
```

Out[170]:

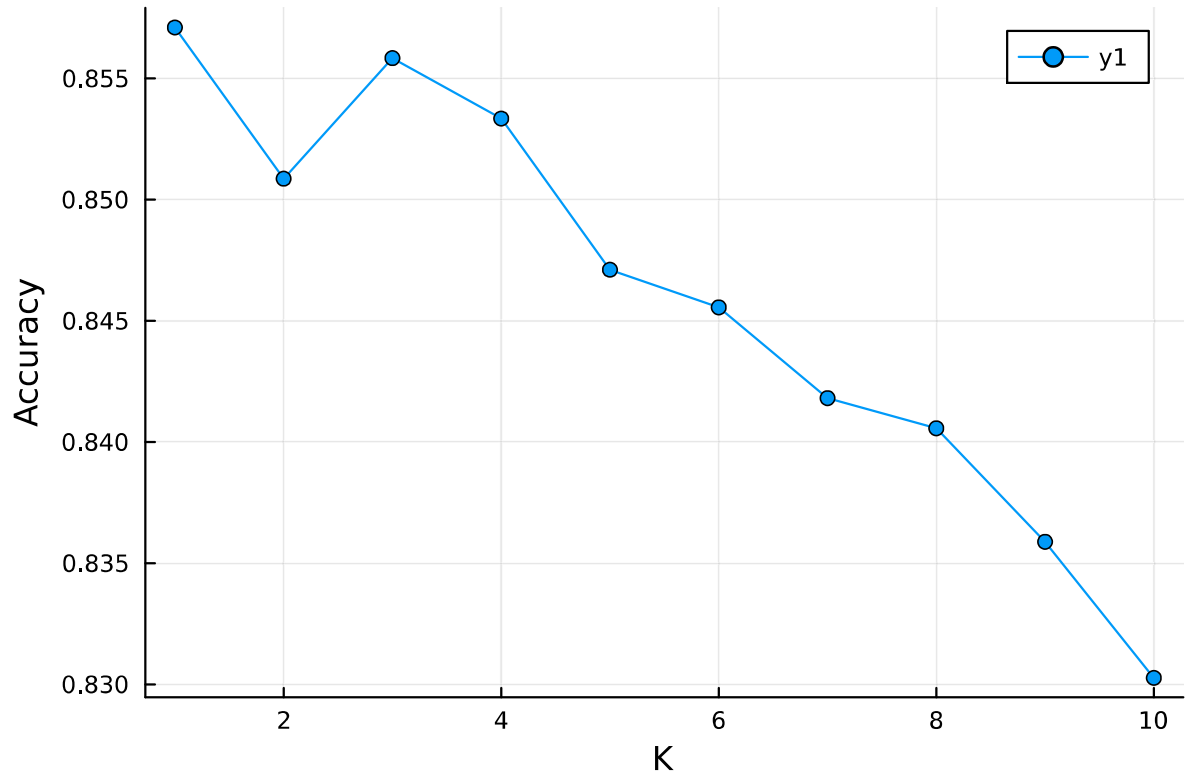


In [171…

```
plot(1:10, vcat(perftrknn1.measurement,perftrknn2.measurement,perftrknn3.measurement,perftrknn4.measurement,perftrknn5.

# Choose k=1 since it has the highest accuracy rate
```

Out[171]:



In [59]:

```
# Combine the balanced training data with the testing data to do the machine learning classification

# Below code is to combine the balanced training data and the testing data

df=vcat(bdftr,dfte)
```

Out[59]: 25,097 rows × 188 columns (omitted printing of 180 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.0789474	0.0845865
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.11315	0.11315
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.281457	0.284768
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.51505	0.438127
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.709641	0.470852
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.0957854	0.103448
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.454545	0.422925
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.40315	0.401575
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.132159	0.0969163
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.0327613	0.0
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.092511	0.061674
12	0.962963	1.0	0.83878	0.337691	0.0718954	0.246187	0.400871	0.320261
13	0.990826	1.0	0.454128	0.0917431	0.0688073	0.12844	0.0963303	0.0825688
14	0.9631	0.512915	0.568266	0.557196	0.575646	0.546125	0.546125	0.476015
15	1.0	0.947566	0.273408	0.0	0.101124	0.172285	0.11985	0.168539
16	0.986708	0.93865	0.875256	0.803681	0.687117	0.537832	0.380368	0.276074
17	0.915865	0.879808	0.769231	0.610577	0.454327	0.399038	0.372596	0.401442
18	0.929648	0.68593	0.211055	0.0125628	0.00502513	0.0351759	0.0251256	0.00753769
19	0.527964	0.346756	0.313199	0.304251	0.261745	0.239374	0.212528	0.174497
20	0.940994	0.680124	0.13354	0.0869565	0.15528	0.124224	0.0993789	0.0962733
21	0.858456	0.880515	0.931985	1.0	0.985294	0.639706	0.349265	0.235294
22	0.948598	0.831776	0.200935	0.00934579	0.0420561	0.0934579	0.116822	0.130841
23	0.945183	0.877076	0.252492	0.0	0.232558	0.318937	0.277409	0.257475

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.906306	0.836036	0.563964	0.174775	0.0	0.0522523	0.113514	0.158559
25	0.90566	0.733491	0.141509	0.0542453	0.0518868	0.00707547	0.0188679	0.00471698
26	1.0	0.983687	0.807504	0.525285	0.336052	0.24633	0.17292	0.153344
27	0.975664	0.809735	0.143805	0.123894	0.265487	0.265487	0.223451	0.238938
28	1.0	0.93913	0.898551	0.75942	0.652174	0.602899	0.536232	0.513043
29	0.620112	0.586592	0.416201	0.150838	0.0	0.0586592	0.153631	0.192737
30	0.901449	0.692754	0.542029	0.321739	0.191304	0.115942	0.101449	0.0637681
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [60]:
```
# Seperate and create y (response variable) and X (all the features)

y, X = unpack(df, ==(:Column188),name->true)
```

Out[60]: ([3.0, 2.0, 2.0, 4.0, 2.0, 1.0, 4.0, 2.0, 4.0, 4.0 … 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0, 4.0], **25097×187 DataFrame**

Row	Column1	Column2	Column3	Column4	Column5	Column6	Colum ⋯
	Float64	Float64	Float64	Float64	Float64	Float64	Float ⋯
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.078 ⋯
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.113
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.281
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.515
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.709 ⋯
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.095
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.454
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.403
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.132 ⋯
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.032
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.092
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮ ⋱
25088	0.862	0.613	0.651	0.647	0.632	0.58	0.509
25089	1.0	0.464	0.481	0.506	0.575	0.592	0.644 ⋯
25090	0.993	0.542	0.542	0.553	0.549	0.525	0.528
25091	1.0	0.93	0.858	0.774	0.658	0.525	0.4
25092	0.802	0.524	0.518	0.473	0.454	0.409	0.369
25093	0.929	0.871	0.805	0.743	0.651	0.536	0.394 ⋯
25094	0.803	0.692	0.587	0.447	0.318	0.19	0.118
25095	1.0	0.967	0.62	0.347	0.139	0.089	0.104
25096	0.984	0.567	0.607	0.583	0.607	0.575	0.575
25097	0.974	0.913	0.866	0.823	0.746	0.642	0.548 ⋯

181 columns and 25076 rows omitted, **0×0 DataFrame**)

In [61]:
```
y
```

Out[61]:
```
25097-element Vector{Float64}:
 3.0
 2.0
 2.0
 4.0
 2.0
 1.0
 4.0
 2.0
 4.0
 4.0
 3.0
 2.0
 0.0
 ⋮
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
```

In [62]:
```
# Set y to be categorical for MLJ

y = CategoricalArray(y)
```

Out[62]:
```
25097-element CategoricalArray{Float64,1,UInt32}:
 3.0
 2.0
 2.0
 4.0
 2.0
 1.0
 4.0
 2.0
 4.0
 4.0
 3.0
 2.0
 0.0
 ⋮
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
```

In [63]:
```
X
```

Out[63]: 25,097 rows × 187 columns (omitted printing of 179 columns)

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
1	1.0	0.951128	0.449248	0.203008	0.12406	0.114662	0.0789474	0.0845865
2	0.501529	0.474006	0.330275	0.0428135	0.0	0.0703364	0.11315	0.11315
3	0.0	0.0281457	0.157285	0.327815	0.438742	0.35596	0.281457	0.284768
4	1.0	0.525084	0.545151	0.545151	0.551839	0.521739	0.51505	0.438127
5	1.0	0.994395	0.936099	0.911435	0.912556	0.840807	0.709641	0.470852
6	0.796935	0.704981	0.180077	0.0	0.0536398	0.0957854	0.0957854	0.103448
7	1.0	0.478261	0.490119	0.494071	0.482213	0.44664	0.454545	0.422925
8	0.0	0.080315	0.15748	0.225197	0.322835	0.4	0.40315	0.401575
9	0.763583	0.67254	0.56094	0.459618	0.325991	0.211454	0.132159	0.0969163
10	0.736349	0.633385	0.5039	0.368175	0.227769	0.118565	0.0327613	0.0
11	1.0	0.857562	0.600587	0.277533	0.0866373	0.104258	0.092511	0.061674
12	0.962963	1.0	0.83878	0.337691	0.0718954	0.246187	0.400871	0.320261
13	0.990826	1.0	0.454128	0.0917431	0.0688073	0.12844	0.0963303	0.0825688
14	0.9631	0.512915	0.568266	0.557196	0.575646	0.546125	0.546125	0.476015
15	1.0	0.947566	0.273408	0.0	0.101124	0.172285	0.11985	0.168539
16	0.986708	0.93865	0.875256	0.803681	0.687117	0.537832	0.380368	0.276074
17	0.915865	0.879808	0.769231	0.610577	0.454327	0.399038	0.372596	0.401442
18	0.929648	0.68593	0.211055	0.0125628	0.00502513	0.0351759	0.0251256	0.00753769
19	0.527964	0.346756	0.313199	0.304251	0.261745	0.239374	0.212528	0.174497
20	0.940994	0.680124	0.13354	0.0869565	0.15528	0.124224	0.0993789	0.0962733
21	0.858456	0.880515	0.931985	1.0	0.985294	0.639706	0.349265	0.235294
22	0.948598	0.831776	0.200935	0.00934579	0.0420561	0.0934579	0.116822	0.130841
23	0.945183	0.877076	0.252492	0.0	0.232558	0.318937	0.277409	0.257475

	Column1	Column2	Column3	Column4	Column5	Column6	Column7	Column8
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64
24	0.906306	0.836036	0.563964	0.174775	0.0	0.0522523	0.113514	0.158559
25	0.90566	0.733491	0.141509	0.0542453	0.0518868	0.00707547	0.0188679	0.00471698
26	1.0	0.983687	0.807504	0.525285	0.336052	0.24633	0.17292	0.153344
27	0.975664	0.809735	0.143805	0.123894	0.265487	0.265487	0.223451	0.238938
28	1.0	0.93913	0.898551	0.75942	0.652174	0.602899	0.536232	0.513043
29	0.620112	0.586592	0.416201	0.150838	0.0	0.0586592	0.153631	0.192737
30	0.901449	0.692754	0.542029	0.321739	0.191304	0.115942	0.101449	0.0637681
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

In [64]:
```
# Set training data for MLJ

tr_inds=collect(1:size(bXtrdf)[1])
```

Out[64]:
```
3205-element Vector{Int64}:
    1
    2
    3
    4
    5
    6
    7
    8
    9
   10
   11
   12
   13
    ⋮
 3194
 3195
 3196
 3197
 3198
 3199
 3200
 3201
 3202
 3203
 3204
 3205
```

In [65]:
```
# Set testing data for MLJ

te_inds=collect(size(bXtrdf)[1]+1:size(X)[1])
```

Out[65]:
```
21892-element Vector{Int64}:
  3206
  3207
  3208
  3209
  3210
  3211
  3212
  3213
  3214
  3215
  3216
  3217
  3218
     ⋮
 25086
 25087
 25088
 25089
 25090
 25091
 25092
 25093
 25094
 25095
 25096
 25097
```

In [66]:
```
# Build KNN model for best K  when k=1

knnc = KNNClassifier(K=1)
knnc_mach = machine(knnc, X, y)
```

Out[66]:
```
untrained Machine; caches model-specific representations of data
  model: KNNClassifier(K = 1, …)
  args:
    1:  Source @072 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @154 ⏎ AbstractVector{Multiclass{5}}
```

In [67]:
```
MLJ.fit!(knnc_mach, rows=tr_inds)
```

```
┌ Info: Training machine(KNNClassifier(K = 1, …), …).
└ @ MLJBase C:\Users\zizhe\.julia\packages\MLJBase\CtxrQ\src\machines.jl:496
```

Out[67]:
```
trained Machine; caches model-specific representations of data
  model: KNNClassifier(K = 1, …)
  args:
    1:  Source @072 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @154 ⏎ AbstractVector{Multiclass{5}}
```

In [68]:
```
# Test data classification: predict testing data classification using KNN, when k=1

ypredknn=predict_mode(knnc_mach, rows=te_inds)
```

Out[68]:
```
21892-element CategoricalArray{Float64,1,UInt32}:
 0.0
 1.0
 0.0
 0.0
 4.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 3.0
 ⋮
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
```

In [69]:
```
# Performance evaluation for KNN, when k=1
```

In [70]:
```
#1. Accuracy rate of the test data for KNN

accuracy(ypredknn, y[te_inds])
```

Out[70]:
```
0.7795084962543395
```

In [ ]:
```
# We can find the prediction accuracy rate is 0.7995. This means there is 77.95% correctly predict the heartbeat type i
```

In [71]:
```
# 2. Confusion matrix for KNN

CMKNN=ConfusionMatrix()(ypredknn, y[te_inds])
```

```
┌ Warning: The classes are un-ordered,
│ using order: [0.0, 1.0, 2.0, 3.0, 4.0].
│ To suppress this warning, consider coercing to OrderedFactor.
└ @ MLJBase C:\Users\zizhe\.julia\packages\MLJBase\CtxrQ\src\measures\confusion_matrix.jl:122
```

Out[71]:
```
5×5 Matrix{Int64}:
 13684   73    53    4    25
  2132  440    44    1    10
   837   22  1269    9    32
  1195   18    69  148    17
   270    3    13    0  1524
```

In [ ]:
```
# From the confusion matrix, we can see that apart from the correct classification, the misclassification majority happ

# But majority classification was predicted correctly, e.g. in class 3, 148 numbers was classified as class 3. 4 number
```

In [ ]:
```
# The following are the prediction accuracy rate for each of the 5 heartbeat types.
```

In [72]:
```
println("Category N prediction accuracy rate by using KNN is ", join([round(CMKNN[1,1]/sum(CMKNN[:,1]); digits=4)*100,"
```

```
Category N prediction accuracy rate by using KNN is 75.53%
```

In [73]:
```
println("Category S prediction accuracy rate by using KNN is ", join([round(CMKNN[2,2]/sum(CMKNN[:,2]); digits=4)*100,"
```

```
Category S prediction accuracy rate by using KNN is 79.14%
```

In [74]:
```
println("Category V prediction accuracy rate by using KNN is ", join([round(CMKNN[3,3]/sum(CMKNN[:,3]); digits=4)*100,"
```

```
Category V prediction accuracy rate by using KNN is 87.64%
```

In [75]:
```
println("Category F prediction accuracy rate by using KNN is ", join([round(CMKNN[4,4]/sum(CMKNN[:,4]); digits=4)*100,"
```

```
Category F prediction accuracy rate by using KNN is 91.36%
```

In [76]:
```
println("Category Q prediction accuracy rate by using KNN is ", join([round(CMKNN[5,5]/sum(CMKNN[:,5]); digits=4)*100,"
```

```
Category Q prediction accuracy rate by using KNN is 94.78%
```

In [ ]:
```
# We can find that the prediction accuracy rate of different heartbeat types varies greatly,
# The reason may be that Category Q, Category F and Category V are very different from other heartbeat categories,
# so it is easy to identify Category Q and Category F and Category V ,
# so the prediction accuracy of Category Q and Category F and Category V is relatively high.
# Since the Category N and Category S are similar, these two categories are easy to be confused,
# so the prediction accuracy of these two categories is not as high as the last three categories Q, F and V.
```

In [ ]:
```
```

In [77]:
```
# Machine learning classification algorithm 2.Decision tree -- no need feature extraction, quick
```

In [78]:
```
import Pkg; Pkg.add("DecisionTree")
Pkg.add("MLJDecisionTreeInterface")

Tree = @load DecisionTreeClassifier pkg=DecisionTree
import MLJDecisionTreeInterface
```

```
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
┌ Info: For silent loading, specify `verbosity=0`.
└ @ Main C:\Users\zizhe\.julia\packages\MLJModels\K5pPR\src\loading.jl:159
import MLJDecisionTreeInterface ✓
```

In [79]:
```
# Performance tuning: use 10 folds cross validation to build models using tree depth = 1 to 10

# To tune the model and select the best hyperparameters that can create the model which has the highest accuracy
```

In [80]:
```
Random.seed!(13)

cv=CV(nfolds=10)

perftrdt1 = evaluate!(machine(Tree(max_depth=1), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt2 = evaluate!(machine(Tree(max_depth=2), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt3 = evaluate!(machine(Tree(max_depth=3), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt4 = evaluate!(machine(Tree(max_depth=4), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt5 = evaluate!(machine(Tree(max_depth=5), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt6 = evaluate!(machine(Tree(max_depth=6), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt7 = evaluate!(machine(Tree(max_depth=7), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt8 = evaluate!(machine(Tree(max_depth=8), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
```

```
perftrdt9 = evaluate!(machine(Tree(max_depth=9), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
perftrdt10 = evaluate!(machine(Tree(max_depth=10), bXtrdf, bytrdf), measure=accuracy,resampling=cv, verbosity=0)
```

Out[80]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict_mode	0.809	0.0114	[0.819, 0.807, 0.816, 0. ⋯

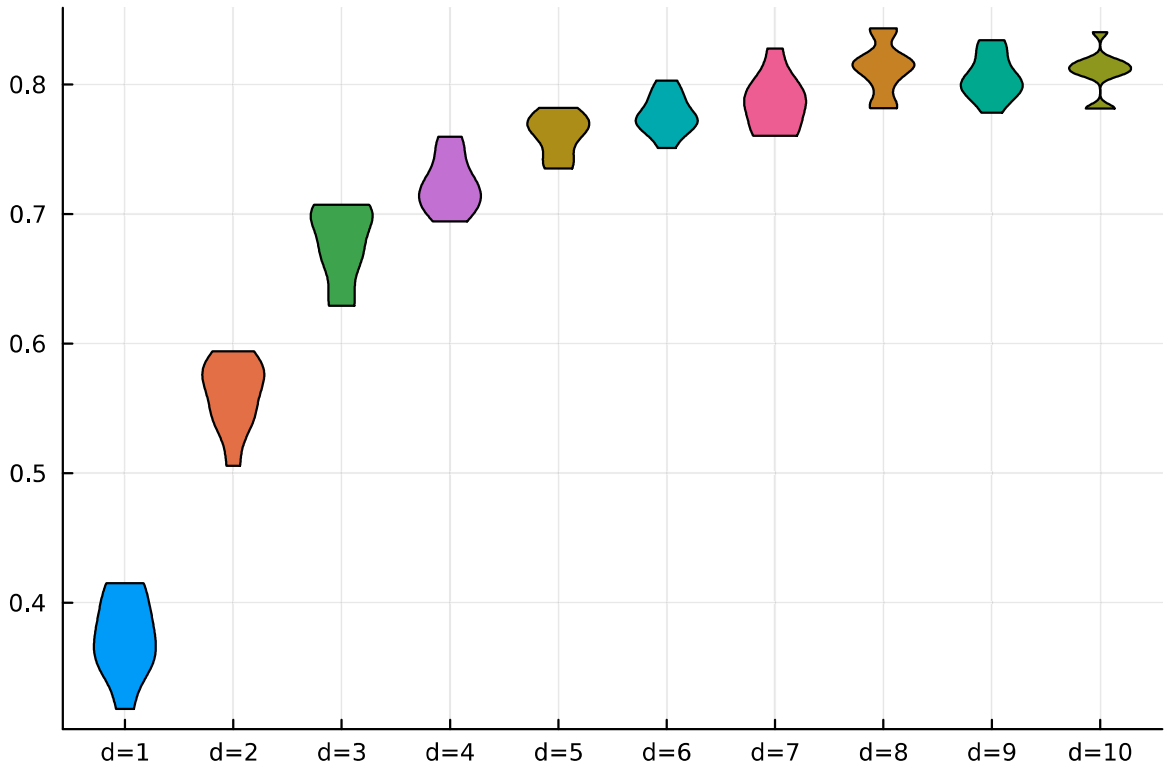
1 column omitted

In [81]:

```
# Plot the 10 folds cross validation accuracy

violin(["d=1"], perftrdt1.per_fold, label=nothing)
violin!(["d=2"], perftrdt2.per_fold, label=nothing)
violin!(["d=3"], perftrdt3.per_fold, label=nothing)
violin!(["d=4"], perftrdt4.per_fold, label=nothing)
violin!(["d=5"], perftrdt5.per_fold, label=nothing)
violin!(["d=6"], perftrdt6.per_fold, label=nothing)
violin!(["d=7"], perftrdt7.per_fold, label=nothing)
violin!(["d=8"], perftrdt8.per_fold, label=nothing)
violin!(["d=9"], perftrdt9.per_fold, label=nothing)
violin!(["d=10"], perftrdt10.per_fold, label=nothing)
```

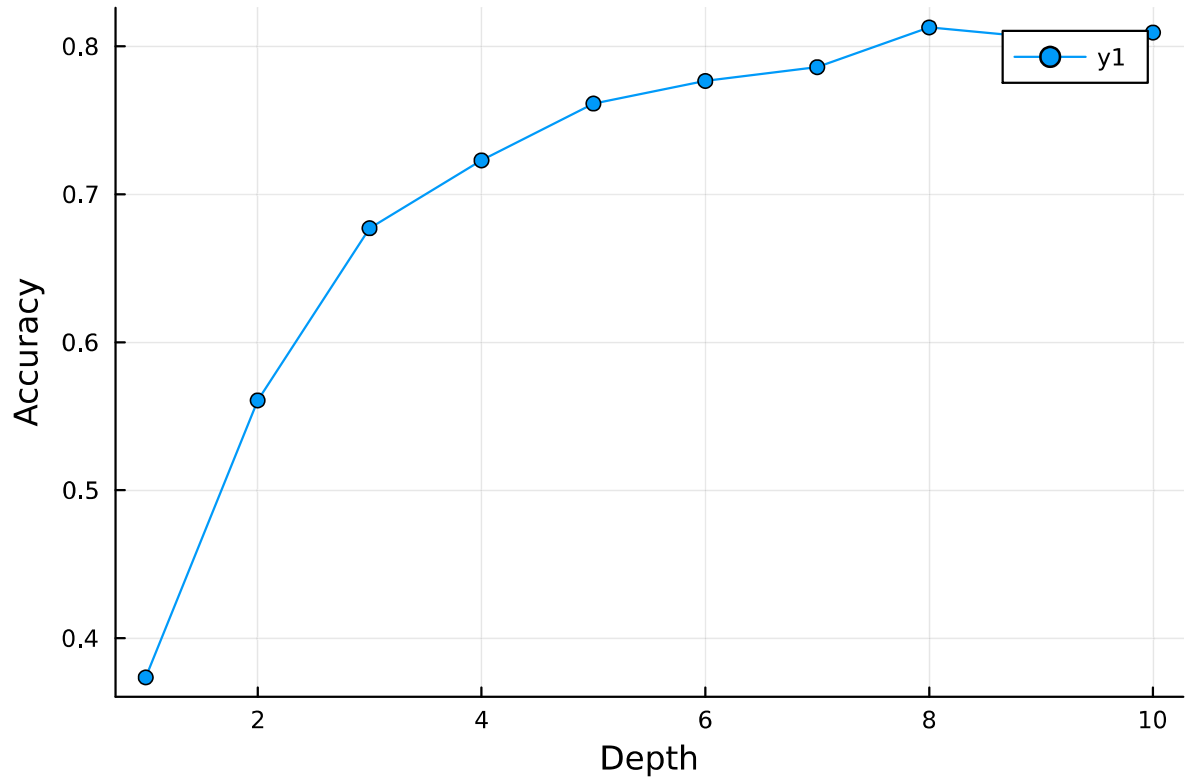
Out[81]:



In [82]:
```
plot(1:10, vcat(perftrdt1.measurement,perftrdt2.measurement,perftrdt3.measurement,perftrdt4.measurement,perftrdt5.measu

# Choose depth = 8 since the accuracy rate is highest, and after that the accuracy rate starts decreasing
```

Out[82]:



In [83]:

```
# Build decision tree model for best depth, which the best depth is depth=8

using MLJ

tree = Tree(max_depth=8)
mach = machine(tree, X, y)
MLJ.fit!(mach, rows=tr_inds)
```

```
┌ Info: Training machine(DecisionTreeClassifier(max_depth = 8, …), …).
└ @ MLJBase C:\Users\zizhe\.julia\packages\MLJBase\CtxrQ\src\machines.jl:496
```

Out[83]:

```
trained Machine; caches model-specific representations of data
  model: DecisionTreeClassifier(max_depth = 8, …)
  args:
    1:  Source @104 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @302 ⏎ AbstractVector{Multiclass{5}}
```

In [84]:

```
# Test data classification: predict testing data classification using decision tree, when depth=8

ypreddt = predict_mode(mach, rows=te_inds)
```

Out[84]:
```
21892-element CategoricalArray{Float64,1,UInt32}:
 0.0
 3.0
 0.0
 0.0
 4.0
 2.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 3.0
 ⋮
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
 4.0
```

In [85]:
```
# Performance evaluation
```

In [86]:
```
# 1. Accuracy rate of the test data by using decision tree, when depth=8

accuracy(ypreddt, y[te_inds])
```

Out[86]:
```
0.7507765393751142
```

In [ ]:
```
# We can find the prediction accuracy rate is 0.7507. This means there is 75.07% correctly predict the heartbeat type i
```

In [88]:
```
# 2.Confusion matrix for decision tree

CMDT=ConfusionMatrix()(ypreddt,y[te_inds])
```

```
┌ Warning: The classes are un-ordered,
│ using order: [0.0, 1.0, 2.0, 3.0, 4.0].
│ To suppress this warning, consider coercing to OrderedFactor.
└ @ MLJBase C:\Users\zizhe\.julia\packages\MLJBase\CtxrQ\src\measures\confusion_matrix.jl:122
```

Out[88]:
```
5×5 Matrix{Int64}:
 13246  124    62    5    48
  1376  375    25    5    17
  1459   24  1209   15    46
  1446   27    92  136    27
   591    6    60    1  1470
```

In [ ]:
```
# Use the heartbeat type 2 as an example to explain the confusion matrix result

# In the class 2 column
# 1209 numbers was classified as class 2.
# 62 numbers was classified as class 0.
# 25 numbers was classified as class 1.
# 92 numbers was classified as class 3.
# 60 numbers was classified as class 4.
```

In [ ]:
```
# The following are the prediction accuracy rate for each of the 5 heartbeat types.
```

In [89]:
```
println("Category N prediction accuracy rate by using Decision Tree is ", join([round(CMDT[1,1]/sum(CMDT[:,1]); digits=
```

```
Category N prediction accuracy rate by using Decision Tree is 73.11%
```

In [90]:
```
println("Category S prediction accuracy rate by using Decision Tree is ", join([round(CMDT[2,2]/sum(CMDT[:,2]); digits=
```

```
Category S prediction accuracy rate by using Decision Tree is 67.45%
```

In [91]:
```
println("Category V prediction accuracy rate by using Decision Tree is ", join([round(CMDT[3,3]/sum(CMDT[:,3]); digits=
```

```
Category V prediction accuracy rate by using Decision Tree is 83.49%
```

In [92]:
```
println("Category F prediction accuracy rate by using Decision Tree is ", join([round(CMDT[4,4]/sum(CMDT[:,4]); digits=
```

```
Category F prediction accuracy rate by using Decision Tree is 83.95%
```

In [93]:
```
println("Category Q prediction accuracy rate by using Decision Tree is ", join([round(CMDT[5,5]/sum(CMDT[:,5]); digits=
```

```
Category Q prediction accuracy rate by using Decision Tree is 91.42%
```

In [ ]:
```
# We can find that the prediction accuracy rate of different heartbeat types varies greatly,
# The reason may be that Category Q, Category F and Category V are very different from other heartbeat categories,
# so it is easy to identify Category Q and Category F and Category V ,
```

```
# so the prediction accuracy of Category Q and Category F and Category V is relatively high.
# Since Category N and Category S are similar, these two categories are easy to be confused,
# so the prediction accuracy of these two categories is not as high as the last three categories Q, F and V.
```

In [ ]:

In [94]:
```
# Machine learning classification algorithm 3 : Support Vector Machine (SVM) -- need feature extraction
```

In [95]:
```
# SVM needs feature extraction, because if using all the features, the speed to do classification is too slow

# As discovered from the explorary data analysis, the different heartbeat type has different length, timing and amplitu

# So I was thinking to get the RR interval length, QRS interval length, and their amplitudes as the features.

# After tried a number of ways,I didn't success to get the RR interval length, QRS interval length, and their amplitude

# I finally decide to use some basic statistic features for the experiment include: Cluster number, Mean, Std, Max valu

# Max value: one R wave amplitude
# Max column: one R wave timing
```

In [96]:
```
# For the training dataset, Convert data format from dataframe to matrix, otherwise cannot use mean(), std() and etc

# X is the combination of balanced training data plus the testing data

Xm = Matrix(X)
```

Out[96]:
```
25097×187 Matrix{Float64}:
 1.0       0.951128   0.449248  0.203008   …  0.0       0.0       0.0
 0.501529  0.474006   0.330275  0.0428135     0.278287  0.278287  0.29052
 0.0       0.0281457  0.157285  0.327815      0.0       0.0       0.0
 1.0       0.525084   0.545151  0.545151      0.0       0.0       0.0
 1.0       0.994395   0.936099  0.911435      0.0       0.0       0.0
 0.796935  0.704981   0.180077  0.0        …  0.291188  0.298851  0.310345
 1.0       0.478261   0.490119  0.494071      0.0       0.0       0.0
 0.0       0.080315   0.15748   0.225197      0.0       0.0       0.0
 0.763583  0.67254    0.56094   0.459618      0.0       0.0       0.0
 0.736349  0.633385   0.5039    0.368175      0.0       0.0       0.0
 1.0       0.857562   0.600587  0.277533   …  0.0       0.0       0.0
 0.962963  1.0        0.83878   0.337691      0.0       0.0       0.0
 0.990826  1.0        0.454128  0.0917431     0.0       0.0       0.0
 ⋮                                       ⋱              ⋮
 0.892     0.502      0.539     0.553      …  0.0       0.0       0.0
 0.856     0.772      0.682     0.596         0.0       0.0       0.0
 0.862     0.613      0.651     0.647         0.0       0.0       0.0
 1.0       0.464      0.481     0.506         0.0       0.0       0.0
 0.993     0.542      0.542     0.553         0.0       0.0       0.0
 1.0       0.93       0.858     0.774      …  0.0       0.0       0.0
 0.802     0.524      0.518     0.473         0.0       0.0       0.0
 0.929     0.871      0.805     0.743         0.0       0.0       0.0
 0.803     0.692      0.587     0.447         0.0       0.0       0.0
 1.0       0.967      0.62      0.347         0.0       0.0       0.0
 0.984     0.567      0.607     0.583      …  0.0       0.0       0.0
 0.974     0.913      0.866     0.823         0.0       0.0       0.0
```

In [ ]:
```
# Feature extraction, the features which I want to extract include:  Mean, Std, Max value, Max column, Cluster number
```

In [97]:
```
# Get the Standard Deviation of each row, as one feature to use
# dims=2 is to calculate the std of each row, dims=1 is to calculate the std of each column

sd=std(Xm,dims=2)
```

Out[97]:
```
25097×1 Matrix{Float64}:
 0.18597119919319055
 0.14295331803464498
 0.18366271151961414
 0.18485755362483758
 0.17269275419005875
 0.07959819641545748
 0.17211985604139327
 0.24453394059026093
 0.3321583033999098
 0.31440203385591614
 0.18352360573743345
 0.22574666314629602
 0.15834486016202265
 ⋮
 0.16676894596086667
 0.23090928898207963
 0.21078659625160032
 0.1943346630800459
 0.1795947526571715
 0.25969083898379525
 0.2322025770123466
 0.30020741409145085
 0.33044266056822713
 0.1490128418730983
 0.19160803164634432
 0.26379068775801456
```

In [98]:
```
# Get the Mean of each row, as one feature to use

m=mean(Xm,dims=2)
```

Out[98]:
```
25097×1 Matrix{Float64}:
 0.08738088559727857
 0.32167328908083276
 0.22547544026000296
 0.18181818333220354
 0.28900386021537255
 0.304997229221511
 0.13679693998858572
 0.22222409353536718
 0.37305943497561195
 0.3592398244548928
 0.11079962563546583
 0.33623431706651646
 0.0665750871038931
 ⋮
 0.1495208556149733
 0.23443807486631027
 0.21725454545454567
 0.1273845989304812
 0.17278609625668453
 0.2736459893048128
 0.2427668449197861
 0.3303786096256686
 0.3949336898395724
 0.10635262032085556
 0.1898401069518717
 0.28812994652406415
```

In [99]:
```
# Find the max value and max column index for each row
# Findmax is to find the max value and the correponding max value index

findmax(Xm,dims=2)
```

Out[99]:
```
([1.0; 1.0; … ; 1.0; 1.0;;], CartesianIndex{2}[CartesianIndex(1, 1); CartesianIndex(2, 166); … ; CartesianIndex(25096, 100); CartesianIndex(25097, 102);;])
```

In [100…
```
# Get the max value of each row, as one feature to use

maxv=findmax(Xm,dims=2)[1]
```

Out[100]:
```
25097×1 Matrix{Float64}:
 1.0
 1.0
 1.0
 1.0
 1.0
 0.7969348430633545
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 ⋮
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
 1.0
```

In [101…
```
# Get the max column index of each row, as one feature to use

maxc=getindex.(findmax(Xm,dims=2)[2], 2)
```

Out[101]:
```
25097×1 Matrix{Int64}:
   1
 166
 125
   1
   1
   1
   1
  79
  98
 101
   1
   2
   2
   ⋮
  99
 102
  99
   1
 101
   1
  98
 100
 101
   1
 100
 102
```

In [102…
```
# kmeans clustering, use the cluster number as a feature

km1=kmeans(Xm', 5; maxiter=100)

a1 = assignments(km1) # Get the cluster number of each sample
```

Out[102]:
```
25097-element Vector{Int64}:
 4
 5
 5
 5
 5
 5
 5
 3
 1
 2
 2
 4
 5
 3
 ⋮
 3
 5
 5
 3
 5
 5
 5
 2
 2
 3
 5
 5
```

In [103…
```
# Feature selection, evaluate feature importancy by using classifier,see what features contribute the most to the class
```

In [104…
```
# Classification process need the data format to be as dataframe, so convert data format to dataframe

# Combine the feature cluster number and the feature standard deviation

X1df=DataFrame(hcat(a1,sd),:auto)
```

Out[104]: 25,097 rows × 2 columns

	x1	x2
	Float64	Float64
1	4.0	0.185971
2	5.0	0.142953
3	5.0	0.183663
4	5.0	0.184858
5	5.0	0.172693
6	5.0	0.0795982
7	3.0	0.17212
8	1.0	0.244534
9	2.0	0.332158
10	2.0	0.314402
11	4.0	0.183524
12	5.0	0.225747
13	3.0	0.158345
14	5.0	0.173819
15	5.0	0.186602
16	2.0	0.291503
17	3.0	0.163309
18	3.0	0.143624
19	5.0	0.176383
20	3.0	0.131044
21	4.0	0.223941
22	5.0	0.202858
23	5.0	0.170457

	x1	x2
	Float64	Float64
24	4.0	0.227136
25	4.0	0.201689
26	3.0	0.133603
27	3.0	0.173206
28	5.0	0.219783
29	5.0	0.221904
30	4.0	0.21231
⋮	⋮	⋮

In [ ]:
```
# Evaluate the above two features by using SVM classifier
```

In [105…
```
import Pkg; Pkg.add("MLJLIBSVMInterface")
import Pkg; Pkg.add("LIBSVM")

SVC = @load SVC pkg=LIBSVM
@load SVC pkg=LIBSVM;

using LIBSVM
```

```
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
   Resolving package versions...
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Project.toml`
  No Changes to `C:\Users\zizhe\.julia\environments\v1.7\Manifest.toml`
┌ Info: For silent loading, specify `verbosity=0`.
└ @ Main C:\Users\zizhe\.julia\packages\MLJModels\K5pPR\src\loading.jl:159
import MLJLIBSVMInterface ✓
import MLJLIBSVMInterface ✓
┌ Info: For silent loading, specify `verbosity=0`.
└ @ Main C:\Users\zizhe\.julia\packages\MLJModels\K5pPR\src\loading.jl:159
WARNING: using LIBSVM.SVC in module Main conflicts with an existing identifier.
```

In [106…
```
# Assume the degree = 1 to assess the importance of all the features extracted
```

```
# SVC can handle multi classification

svc_model = SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1));

svc_mach = machine(svc_model, X1df, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[106]:

```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @931 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @919 ⏎ AbstractVector{Multiclass{5}}
```

In [107…

```
# Evaluate the accuracy

Random.seed!(5)

X1p1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), X1df[tr_inds,:], y[tr_inds]), measure=a
```

Out[107]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.306	0.0229	[0.305, 0.265, 0.318, 0.271 ⋯

1 column omitted

In [108…

```
# Combine the features cluster number,standard deviation, mean. Evaluate these features by using SVM classfier

X2df=DataFrame(hcat(a1,sd,m),:auto)
```

Out[108]: 25,097 rows × 3 columns

	x1	x2	x3
	Float64	Float64	Float64
1	4.0	0.185971	0.0873809
2	5.0	0.142953	0.321673
3	5.0	0.183663	0.225475
4	5.0	0.184858	0.181818
5	5.0	0.172693	0.289004
6	5.0	0.0795982	0.304997
7	3.0	0.17212	0.136797
8	1.0	0.244534	0.222224
9	2.0	0.332158	0.373059
10	2.0	0.314402	0.35924
11	4.0	0.183524	0.1108
12	5.0	0.225747	0.336234
13	3.0	0.158345	0.0665751
14	5.0	0.173819	0.169327
15	5.0	0.186602	0.295399
16	2.0	0.291503	0.336516
17	3.0	0.163309	0.224355
18	3.0	0.143624	0.0634053
19	5.0	0.176383	0.184522
20	3.0	0.131044	0.0688212
21	4.0	0.223941	0.111464
22	5.0	0.202858	0.247639
23	5.0	0.170457	0.191439

	x1	x2	x3
	Float64	Float64	Float64
24	4.0	0.227136	0.202602
25	4.0	0.201689	0.162912
26	3.0	0.133603	0.171149
27	3.0	0.173206	0.0912285
28	5.0	0.219783	0.234845
29	5.0	0.221904	0.219834
30	4.0	0.21231	0.120949
⋮	⋮	⋮	⋮

In [109…
```
svc_mach = machine(svc_model, X2df, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[109]:
```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @181 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @130 ⏎ AbstractVector{Multiclass{5}}
```

In [110…
```
# Evaluate the accuracy

Random.seed!(6)

X2p1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), X2df[tr_inds,:], y[tr_inds]), measure=a
```

Out[110]:
```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.407	0.0254	[0.445, 0.361, 0.396, 0.449 ⋯

1 column omitted

In [111…

```
# Combine the features cluster number,standard deviation, mean, max value. Evaluate these features by using SVM classfi

X3df=DataFrame(hcat(a1,sd,m,maxv),:auto)
```

Out[111]: 25,097 rows × 4 columns

	x1	x2	x3	x4
	Float64	Float64	Float64	Float64
1	4.0	0.185971	0.0873809	1.0
2	5.0	0.142953	0.321673	1.0
3	5.0	0.183663	0.225475	1.0
4	5.0	0.184858	0.181818	1.0
5	5.0	0.172693	0.289004	1.0
6	5.0	0.0795982	0.304997	0.796935
7	3.0	0.17212	0.136797	1.0
8	1.0	0.244534	0.222224	1.0
9	2.0	0.332158	0.373059	1.0
10	2.0	0.314402	0.35924	1.0
11	4.0	0.183524	0.1108	1.0
12	5.0	0.225747	0.336234	1.0
13	3.0	0.158345	0.0665751	1.0
14	5.0	0.173819	0.169327	1.0
15	5.0	0.186602	0.295399	1.0
16	2.0	0.291503	0.336516	1.0
17	3.0	0.163309	0.224355	1.0
18	3.0	0.143624	0.0634053	1.0
19	5.0	0.176383	0.184522	1.0
20	3.0	0.131044	0.0688212	1.0
21	4.0	0.223941	0.111464	1.0
22	5.0	0.202858	0.247639	1.0
23	5.0	0.170457	0.191439	1.0

	x1	x2	x3	x4
	Float64	Float64	Float64	Float64
24	4.0	0.227136	0.202602	1.0
25	4.0	0.201689	0.162912	1.0
26	3.0	0.133603	0.171149	1.0
27	3.0	0.173206	0.0912285	1.0
28	5.0	0.219783	0.234845	1.0
29	5.0	0.221904	0.219834	1.0
30	4.0	0.21231	0.120949	1.0
⋮	⋮	⋮	⋮	⋮

In [112…

```
svc_mach = machine(svc_model, X3df, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[112]:

```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @957 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @312 ⏎ AbstractVector{Multiclass{5}}
```

In [113…

```
# Evaluate the accuracy

Random.seed!(7)

X3p1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), X3df[tr_inds,:], y[tr_inds]), measure=a
```

Out[113]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.407	0.0249	[0.442, 0.364, 0.399, 0.449 ⋯

1 column omitted

In [114…

```
# Combine the features cluster number,standard deviation, mean, max value,max column index. Evaluate these features by

X4df=DataFrame(hcat(a1,sd,m,maxv,maxc),:auto)
```

Out[114]: 25,097 rows × 5 columns

	x1	x2	x3	x4	x5
	Float64	Float64	Float64	Float64	Float64
1	4.0	0.185971	0.0873809	1.0	1.0
2	5.0	0.142953	0.321673	1.0	166.0
3	5.0	0.183663	0.225475	1.0	125.0
4	5.0	0.184858	0.181818	1.0	1.0
5	5.0	0.172693	0.289004	1.0	1.0
6	5.0	0.0795982	0.304997	0.796935	1.0
7	3.0	0.17212	0.136797	1.0	1.0
8	1.0	0.244534	0.222224	1.0	79.0
9	2.0	0.332158	0.373059	1.0	98.0
10	2.0	0.314402	0.35924	1.0	101.0
11	4.0	0.183524	0.1108	1.0	1.0
12	5.0	0.225747	0.336234	1.0	2.0
13	3.0	0.158345	0.0665751	1.0	2.0
14	5.0	0.173819	0.169327	1.0	103.0
15	5.0	0.186602	0.295399	1.0	1.0
16	2.0	0.291503	0.336516	1.0	104.0
17	3.0	0.163309	0.224355	1.0	136.0
18	3.0	0.143624	0.0634053	1.0	76.0
19	5.0	0.176383	0.184522	1.0	98.0
20	3.0	0.131044	0.0688212	1.0	81.0
21	4.0	0.223941	0.111464	1.0	4.0
22	5.0	0.202858	0.247639	1.0	91.0
23	5.0	0.170457	0.191439	1.0	106.0

	x1	x2	x3	x4	x5
	Float64	Float64	Float64	Float64	Float64
24	4.0	0.227136	0.202602	1.0	83.0
25	4.0	0.201689	0.162912	1.0	79.0
26	3.0	0.133603	0.171149	1.0	1.0
27	3.0	0.173206	0.0912285	1.0	46.0
28	5.0	0.219783	0.234845	1.0	1.0
29	5.0	0.221904	0.219834	1.0	89.0
30	4.0	0.21231	0.120949	1.0	52.0
⋮	⋮	⋮	⋮	⋮	⋮

In [115…

```
svc_mach = machine(svc_model, X4df, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[115]:

```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @578 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @988 ⏎ AbstractVector{Multiclass{5}}
```

In [116…

```
# Evaluate the accuracy

Random.seed!(8)

X4p1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), X4df[tr_inds,:], y[tr_inds]), measure=a
```

Out[116]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.352	0.0193	[0.352, 0.33, 0.374, 0.355, ⋯

1 column omitted

In [117…

```
# So far we know Mean is an important feature, we need to assess if a1 (cluster number) and std are important features
```

In [118…

```
# Combine the features cluster number, mean. Evaluate these features by using SVM classfier

X6df=DataFrame(hcat(a1,m),:auto)
```

Out[118]: 25,097 rows × 2 columns

	x1	x2
	Float64	Float64
1	4.0	0.0873809
2	5.0	0.321673
3	5.0	0.225475
4	5.0	0.181818
5	5.0	0.289004
6	5.0	0.304997
7	3.0	0.136797
8	1.0	0.222224
9	2.0	0.373059
10	2.0	0.35924
11	4.0	0.1108
12	5.0	0.336234
13	3.0	0.0665751
14	5.0	0.169327
15	5.0	0.295399
16	2.0	0.336516
17	3.0	0.224355
18	3.0	0.0634053
19	5.0	0.184522
20	3.0	0.0688212
21	4.0	0.111464
22	5.0	0.247639
23	5.0	0.191439

	x1	x2
	Float64	Float64
24	4.0	0.202602
25	4.0	0.162912
26	3.0	0.171149
27	3.0	0.0912285
28	5.0	0.234845
29	5.0	0.219834
30	4.0	0.120949
⋮	⋮	⋮

In [119…

```
svc_mach = machine(svc_model, X6df, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[119]:

```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @849 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @637 ⏎ AbstractVector{Multiclass{5}}
```

In [120…

```
# Evaluate the accuracy

Random.seed!(9)

X6p1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), X6df[tr_inds,:], y[tr_inds]), measure=a
```

Out[120]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.385	0.017	[0.417, 0.352, 0.374, 0.402 ⋯

1 column omitted

In [121…

```
# Combine the features standard deviation, mean. Evaluate these features by using SVM classfier

X7df=DataFrame(hcat(sd,m),:auto)
```

Out[121]: 25,097 rows × 2 columns

	x1	x2
	Float64	Float64
1	0.185971	0.0873809
2	0.142953	0.321673
3	0.183663	0.225475
4	0.184858	0.181818
5	0.172693	0.289004
6	0.0795982	0.304997
7	0.17212	0.136797
8	0.244534	0.222224
9	0.332158	0.373059
10	0.314402	0.35924
11	0.183524	0.1108
12	0.225747	0.336234
13	0.158345	0.0665751
14	0.173819	0.169327
15	0.186602	0.295399
16	0.291503	0.336516
17	0.163309	0.224355
18	0.143624	0.0634053
19	0.176383	0.184522
20	0.131044	0.0688212
21	0.223941	0.111464
22	0.202858	0.247639
23	0.170457	0.191439

	x1	x2
	Float64	Float64
24	0.227136	0.202602
25	0.201689	0.162912
26	0.133603	0.171149
27	0.173206	0.0912285
28	0.219783	0.234845
29	0.221904	0.219834
30	0.21231	0.120949
⋮	⋮	⋮

In [122…
```
svc_mach = machine(svc_model, X7df, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[122]:
```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @045 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @776 ⏎ AbstractVector{Multiclass{5}}
```

In [123…
```
# Evaluate the accuracy

Random.seed!(10)

X7p1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), X7df[tr_inds,:], y[tr_inds]), measure=a
```

Out[123]:
```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

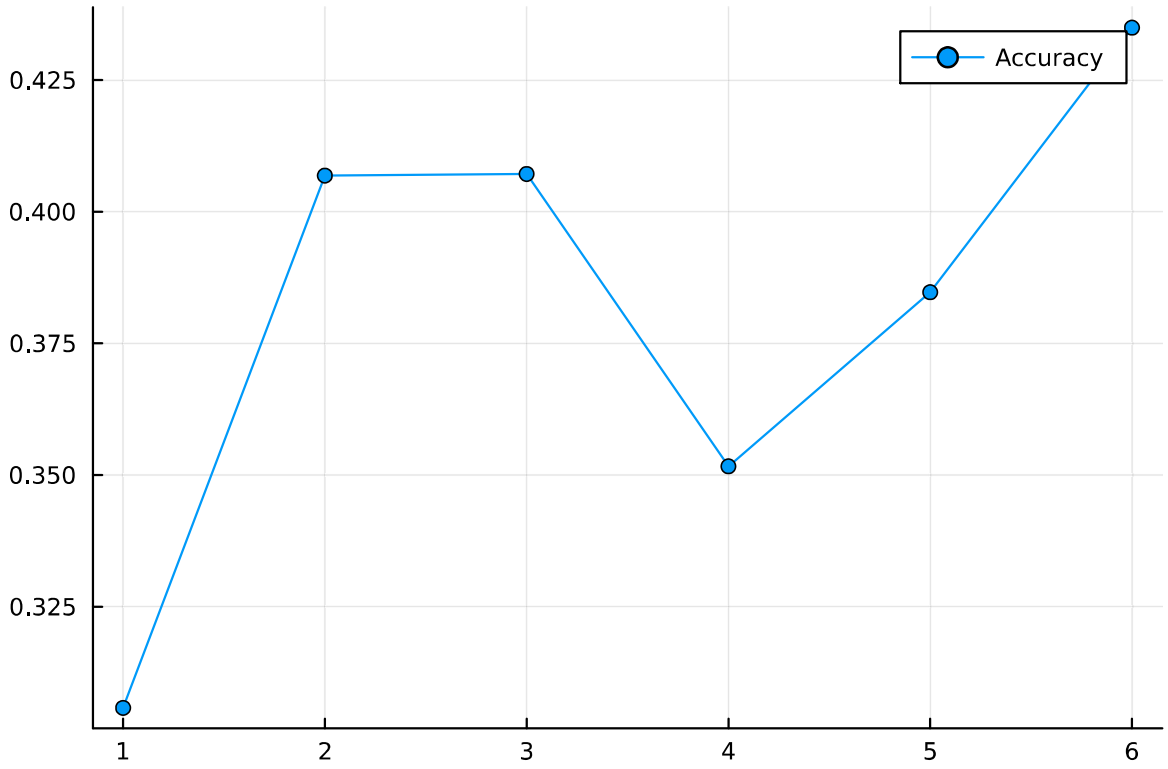
measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.435	0.0133	[0.458, 0.405, 0.433, 0.439 ⋯

1 column omitted

In [124…
```
plot(1:6,vcat(X1p1.measurement,X2p1.measurement,X3p1.measurement,X4p1.measurement,X6p1.measurement,X7p1.measurement),ma

# After tried multiple combinations, I found that the combination of features a1, std and mean has the highest accuracy
```

Out[124]:



In [125…
```
# Combine the features cluster number, standard deviation, mean, these are the final subset of features

Xdf=DataFrame(hcat(a1,sd,m),:auto)
```

Out[125]: 25,097 rows × 3 columns

	x1	x2	x3
	Float64	Float64	Float64
1	4.0	0.185971	0.0873809
2	5.0	0.142953	0.321673
3	5.0	0.183663	0.225475
4	5.0	0.184858	0.181818
5	5.0	0.172693	0.289004
6	5.0	0.0795982	0.304997
7	3.0	0.17212	0.136797
8	1.0	0.244534	0.222224
9	2.0	0.332158	0.373059
10	2.0	0.314402	0.35924
11	4.0	0.183524	0.1108
12	5.0	0.225747	0.336234
13	3.0	0.158345	0.0665751
14	5.0	0.173819	0.169327
15	5.0	0.186602	0.295399
16	2.0	0.291503	0.336516
17	3.0	0.163309	0.224355
18	3.0	0.143624	0.0634053
19	5.0	0.176383	0.184522
20	3.0	0.131044	0.0688212
21	4.0	0.223941	0.111464
22	5.0	0.202858	0.247639
23	5.0	0.170457	0.191439

	x1	x2	x3
	Float64	Float64	Float64
24	4.0	0.227136	0.202602
25	4.0	0.201689	0.162912
26	3.0	0.133603	0.171149
27	3.0	0.173206	0.0912285
28	5.0	0.219783	0.234845
29	5.0	0.221904	0.219834
30	4.0	0.21231	0.120949
⋮	⋮	⋮	⋮

In [126…

```
# Performance tuning: use 10 folds cross validation to build models using degree = 1 to 5

# To tune the model and select the best hyperparameters which can create the model with the highest accuracy

Random.seed!(15)

cv=CV(nfolds=10)
```

Out[126]:

```
CV(
  nfolds = 10,
  shuffle = false,
  rng = Random._GLOBAL_RNG())
```

In [127…

```
# Check the degree from 1 to 5 which one has the highest accuracy on the training data

perftrsvm1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1)), Xdf[tr_inds,:], y[tr_inds]), meas
perftrsvm2 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(2)), Xdf[tr_inds,:], y[tr_inds]), meas
perftrsvm3 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(3)), Xdf[tr_inds,:], y[tr_inds]), meas
perftrsvm4 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(4)), Xdf[tr_inds,:], y[tr_inds]), meas
perftrsvm5 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(5)), Xdf[tr_inds,:], y[tr_inds]), meas
```

Out[127]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.403	0.0204	[0.47, 0.368, 0.383, 0.396, ⋯

1 column omitted

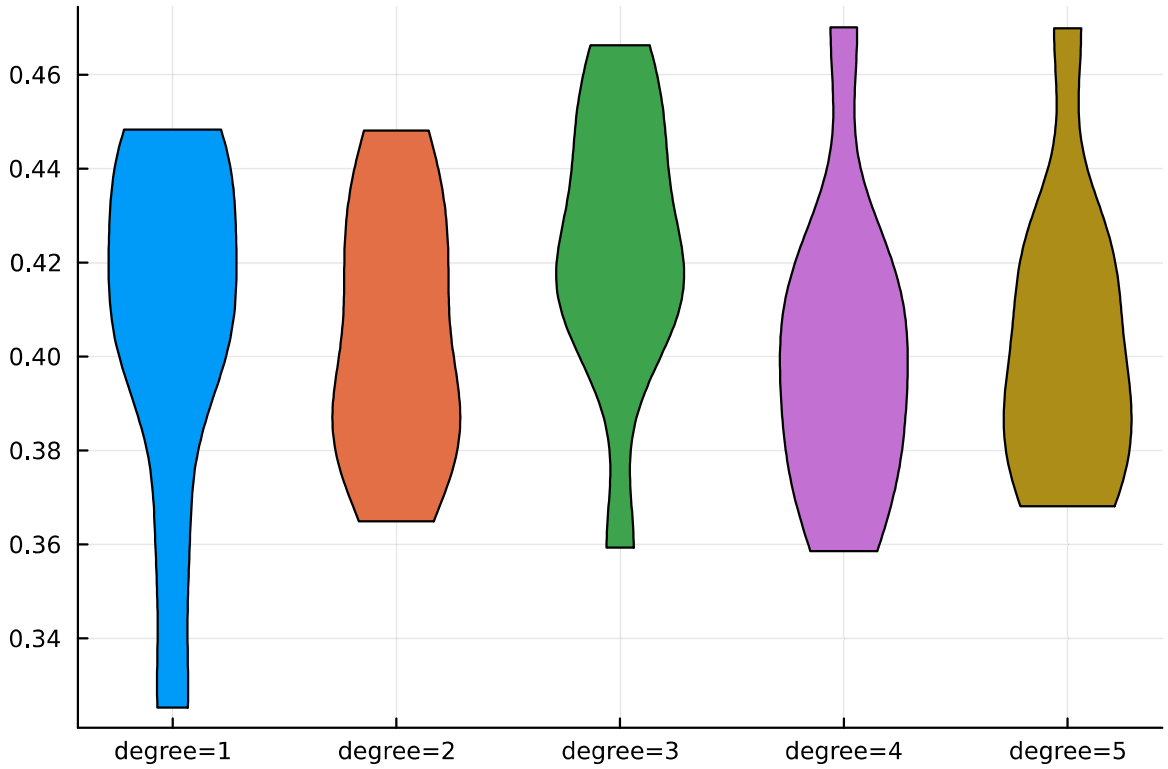
In [128…

```julia
# Plot the degrees

violin(["degree=1"], perftrsvm1.per_fold, label=nothing)
violin!(["degree=2"], perftrsvm2.per_fold, label=nothing)
violin!(["degree=3"], perftrsvm3.per_fold, label=nothing)
violin!(["degree=4"], perftrsvm4.per_fold, label=nothing)
violin!(["degree=5"], perftrsvm5.per_fold, label=nothing)

# Choose degree = 1 as it has the highest accuracy
```

Out[128]:



In [129…

```
# Performance tuning: for degree=1, use 10 folds cross validation to build models, using cost = 1 to 5

# To tune the model and select the best hyperparameters that can create the model which has the highest accuracy

Random.seed!(16)

perftrsvmc1 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1), cost=1.0), Xdf[tr_inds,:], y[tr_i
perftrsvmc2 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1), cost=2.0), Xdf[tr_inds,:], y[tr_i
perftrsvmc3 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1), cost=3.0), Xdf[tr_inds,:], y[tr_i
perftrsvmc4 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1), cost=4.0), Xdf[tr_inds,:], y[tr_i
perftrsvmc5 = evaluate!(machine(SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1), cost=5.0), Xdf[tr_inds,:], y[tr_i
```

Out[129]:

```
PerformanceEvaluation object with these fields:
  measure, operation, measurement, per_fold,
  per_observation, fitted_params_per_fold,
  report_per_fold, train_test_rows
Extract:
```

measure	operation	measurement	1.96*SE	per_fold ⋯
Accuracy()	predict	0.421	0.0129	[0.452, 0.386, 0.405, 0.408 ⋯

1 column omitted

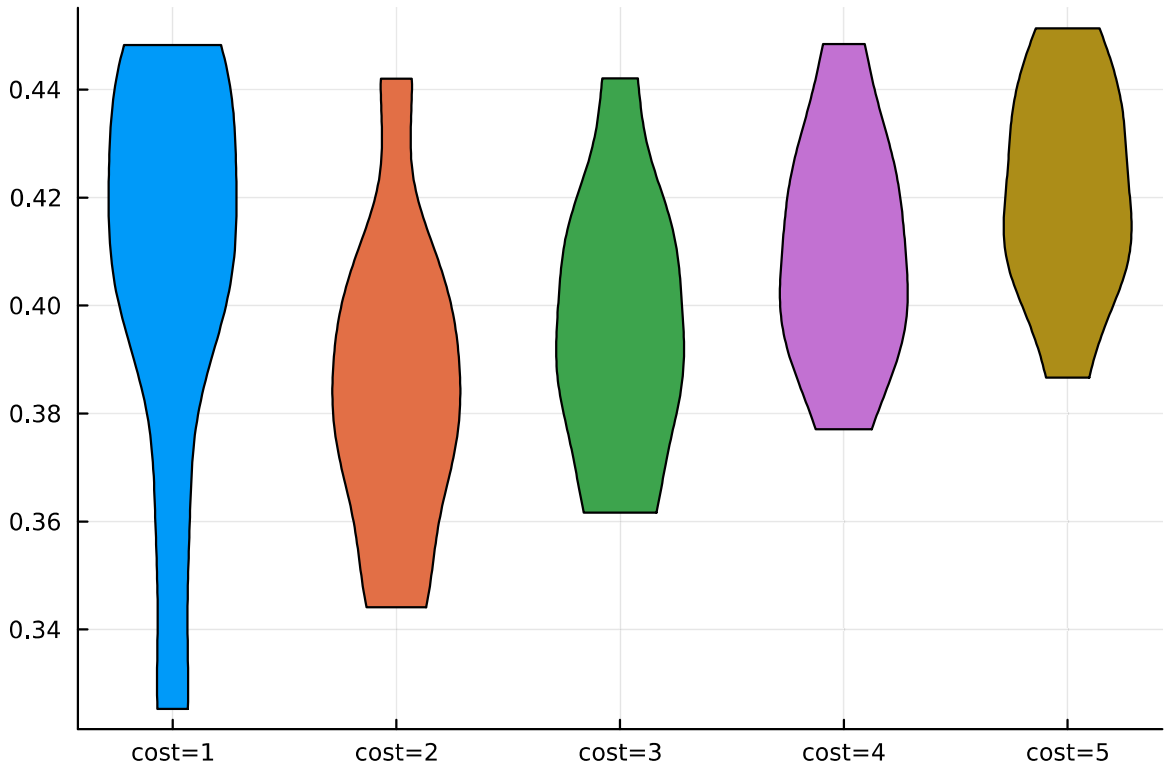
In [130…

```julia
# Plot the cost

violin(["cost=1"], perftrsvmc1.per_fold, label=nothing)
violin!(["cost=2"], perftrsvmc2.per_fold, label=nothing)
violin!(["cost=3"], perftrsvmc3.per_fold, label=nothing)
violin!(["cost=4"], perftrsvmc4.per_fold, label=nothing)
violin!(["cost=5"], perftrsvmc5.per_fold, label=nothing)

# Here I can not see very clear which cost has the highest accuracy, so do another plot to see
```

Out[130]:

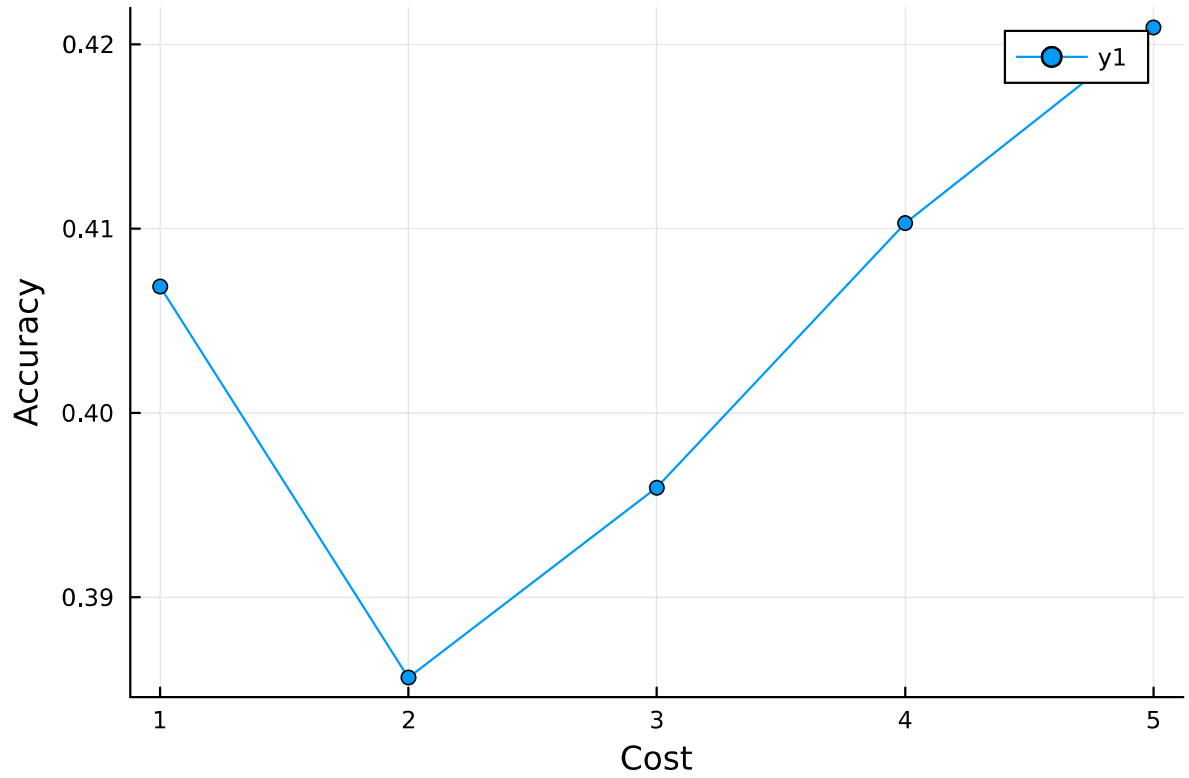


In [132…

```
# Choose degree=1, cost=3, as cost=3 has the highest accuracy rate

plot(1:5, vcat(perftrsvmc1.measurement, perftrsvmc2.measurement,perftrsvmc3.measurement,perftrsvmc4.measurement,perftrs
```

Out[132]:



In [133…

```
# Build SVM classifier after the hyperparameters are decided

svc_model = SVC(kernel=LIBSVM.Kernel.Polynomial, degree=Int32(1),cost=3.0)

svc_mach = machine(svc_model, Xdf, y)

MLJ.fit!(svc_mach, rows=tr_inds, verbosity=0)
```

Out[133]:

```
trained Machine; caches model-specific representations of data
  model: SVC(kernel = Polynomial, …)
  args:
    1:  Source @029 ⏎ Table{AbstractVector{Continuous}}
    2:  Source @628 ⏎ AbstractVector{Multiclass{5}}
```

In [134…

```
# Predict the classification of the test data by using SVM model

ypredsvm = MLJ.predict(svc_mach, rows=te_inds)
```

Out[134]:
```
21892-element CategoricalArray{Float64,1,UInt32}:
 0.0
 0.0
 0.0
 3.0
 3.0
 0.0
 0.0
 1.0
 3.0
 0.0
 0.0
 0.0
 3.0
 ⋮
 0.0
 1.0
 1.0
 0.0
 3.0
 1.0
 1.0
 2.0
 2.0
 0.0
 3.0
 1.0
```

In [135…
```
# Performance evaluation
```

In [136…
```
# 1.Find the predict Accuracy rate of the test data

accuracy(ypredsvm, y[te_inds])
```

Out[136]:
```
0.48131737621048787
```

In [ ]:
```
# We can find the prediction accuracy rate is 0.4813. This means there is 48.13% correctly predict the heartbeat type i
```

In [137…
```
#2. Confusion matrix for SVM

CMSVM=ConfusionMatrix()(ypredsvm, y[te_inds])

# From the confusion matrix, we can see that all classes are not predicting well, for example, in class 1, majority was
# We can mention the %
```

```
┌ Warning: The classes are un-ordered,
│ using order: [0.0, 1.0, 2.0, 3.0, 4.0].
│ To suppress this warning, consider coercing to OrderedFactor.
└ @ MLJBase C:\Users\zizhe\.julia\packages\MLJBase\CtxrQ\src\measures\confusion_matrix.jl:122
```

Out[137]:
```
5×5 Matrix{Int64}:
 9800  174  419   29  365
 2767  190  279    0  424
 1184   31  416    2  589
 4365  161  334  131  230
    2    0    0    0    0
```

In [ ]:
```
# Use the heartbeat type 3 as an example to explain the confusion matrix result

# In the class 3 column
# 131 numbers was classified as class 3.
# 29 numbers was classified as class 0.
# 0 numbers was classified as class 1.
# 2 numbers was classified as class 2.
# 0 numbers was classified as class 4.
```

In [ ]:
```
# The following are the prediction accuracy rate for each of the 5 heartbeat types.
```

In [138…
```
println("Category N prediction accuracy rate by using SVM is ", join([round(CMSVM[1,1]/sum(CMSVM[:,1]); digits=4)*100,"
```

```
Category N prediction accuracy rate by using SVM is 54.09%
```

In [139…
```
println("Category S prediction accuracy rate by using SVM is ", join([round(CMSVM[2,2]/sum(CMSVM[:,2]); digits=4)*100,"
```

```
Category S prediction accuracy rate by using SVM is 34.17%
```

In [140…
```
println("Category V prediction accuracy rate by using SVM is ", join([round(CMSVM[3,3]/sum(CMSVM[:,3]); digits=4)*100,"
```

```
Category V prediction accuracy rate by using SVM is 28.73%
```

In [141…
```
println("Category F prediction accuracy rate by using SVM is ", join([round(CMSVM[4,4]/sum(CMSVM[:,4]); digits=4)*100,"
```

```
Category F prediction accuracy rate by using SVM is 80.86%
```

In [142…
```
println("Category Q prediction accuracy rate by using SVM is ", join([round(CMSVM[5,5]/sum(CMSVM[:,5]); digits=4)*100,"
```

```
Category Q prediction accuracy rate by using SVM is 0.0%
```

In [ ]:
```
# We can find that the prediction accuracy rate of different heartbeat types varies greatly,
# The classification prediction performance of SVM for different heartbeat types is very unstable.
```

```
# For example, the classification prediction accuracy of category Q is 0, and the classification prediction accuracy of
# Compared with KNN and Decision Tree, SVM has the worst ability to classify heartbeat types
```

In [ ]:
```
# svm accuracy is low in this case, because SVM used the features that I extracted, this means SVM highly rely on the f
```

In [ ]:
```

```

In [143…
```

```