CeleraOne test

Solution by Ivan Senilov

The task is to rank features (sensors readings) by importance with respect to ground truth label using given dataset of 400 samples in `.csv` file.

1. Preparation and data outlook

Import all needed libraries:

In [145]:
```
# import data manipulating libs
import pandas as pd
import numpy as np

# import plotting tools
from plotly import tools
from plotly.offline import download_plotlyjs, init_notebook_mode, plot, iplot
import plotly.graph_objs as go
init_notebook_mode(connected=True)
```

Read data to `pandas` DataFrame object, discarding first column as it doesn't contain useful information.

In [146]:
```
data = pd.read_csv("task/task_data.csv").drop(["sample index"], axis=1)
data.head()
```

Out[146]:

	class_label	sensor0	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6	sensor7	sensor8	sensor9
0	1.0	0.834251	0.726081	0.535904	0.214896	0.873788	0.767605	0.111308	0.557526	0.599650	0.665569
1	1.0	0.804059	0.253135	0.869867	0.334285	0.604075	0.494045	0.833575	0.194190	0.014966	0.802918
2	1.0	0.694404	0.595777	0.581294	0.799003	0.762857	0.651393	0.075905	0.007186	0.659633	0.831009
3	1.0	0.783690	0.038780	0.285043	0.627305	0.800620	0.486340	0.827723	0.339807	0.731343	0.892359
4	1.0	0.788835	0.174433	0.348770	0.938244	0.692065	0.377620	0.183760	0.616805	0.492899	0.930969

Let's look at the stats of the data. Looks like all the data has mean around 0.5 and is defined in $[0, 1]$ interval.

In [147]:
```
data.describe()
```

Out[147]:

	class_label	sensor0	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6	sensor7	sensor8	sensor9
count	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000	400.000000
mean	0.000000	0.523661	0.509223	0.481238	0.509752	0.497875	0.501065	0.490480	0.482372	0.482822	0.541933
std	1.001252	0.268194	0.276878	0.287584	0.297712	0.288208	0.287634	0.289954	0.282714	0.296180	0.272490
min	-1.000000	0.007775	0.003865	0.004473	0.001466	0.000250	0.000425	0.000173	0.003322	0.003165	0.000452
25%	-1.000000	0.299792	0.283004	0.235544	0.262697	0.249369	0.269430	0.226687	0.242848	0.213626	0.321264
50%	0.000000	0.534906	0.507583	0.460241	0.510066	0.497842	0.497108	0.477341	0.463438	0.462251	0.578389
75%	1.000000	0.751887	0.727843	0.734937	0.768975	0.743401	0.738854	0.735304	0.732483	0.740542	0.768990
max	1.000000	0.999476	0.998680	0.992963	0.995119	0.999412	0.997367	0.997141	0.998230	0.996098	0.999465

Moreover, the dataset is perfectly balanced:

In [148]:
```
data["class_label"].value_counts()
```

Out[148]:
```
-1.0    200
 1.0    200
Name: class_label, dtype: int64
```

It may be also useful to look at the data in grphical form (plotting two first readings):

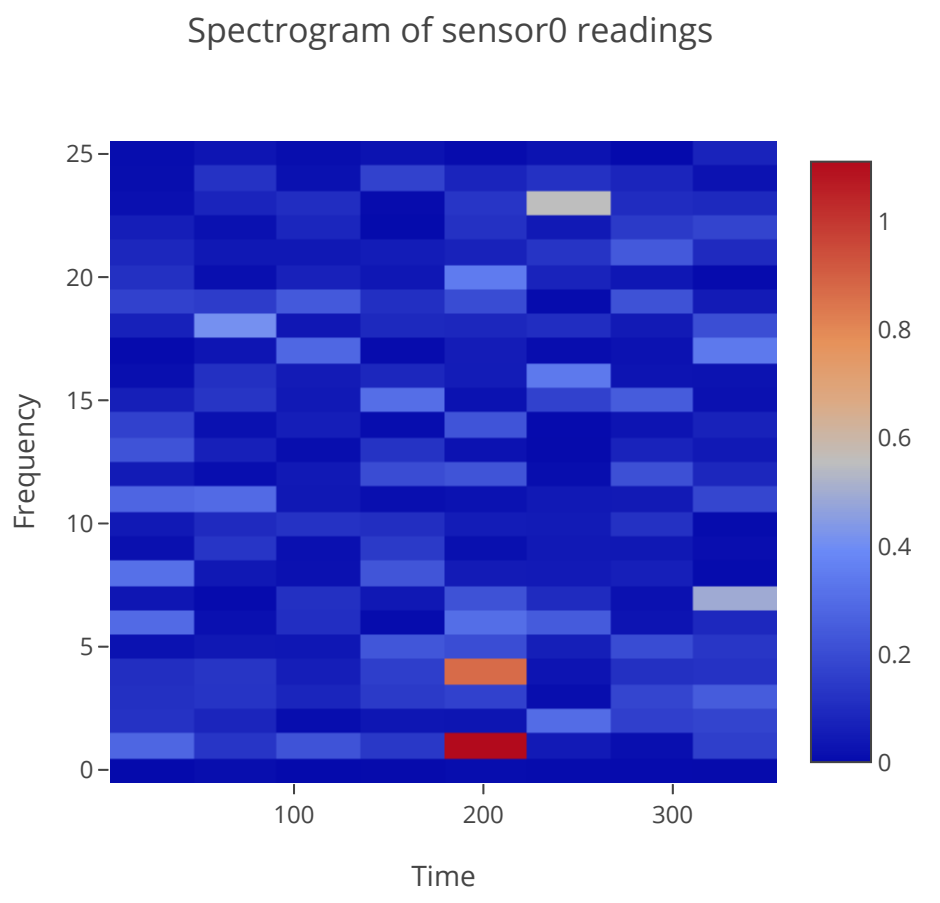
In [149]:
```
iplot(go.Figure(
    data=[go.Scatter(x=data.index, y=data["sensor0"], name="sensor0"),
          go.Scatter(x=data.index, y=data["sensor1"], name="sensor1")],
    layout=go.Layout(title="Raw sensor readings",
                     yaxis=dict(title="Value"),
                     xaxis=dict(title="Measurement"))))
```

Export to plot.ly »

IAdditionally, we may also want to look at the spectrogram of the signal. Moreover, it may be used as feature vector for ML algorithms while flattened or as it is for Convolutional Neural Networks. Let's build the spectrogram with FFT:

In [150]:
```
from scipy.signal import spectrogram
_, t, Sxx = spectrogram(data["sensor0"], nperseg=50)

iplot(go.Figure(
    data=[go.Heatmap(x=t, z=Sxx, name="sensor0")],
    layout=go.Layout(title="Spectrogram of sensor0 readings",
                     yaxis=dict(title="Frequency"),
                     xaxis=dict(title="Time"),
                     width = 500, height = 500)))
```



Export to plot.ly »

Note: it is impossible to evaluate frequencies on the spectrogram as we do not know the sampling frequency.

2. Correlation based features analysis

Let's also look at cross-correlation matrix. We mainly interested in the first column (or first row) that represents correlation between sensor readings and output label:

In [151]:
```
corr_matrix = data.corr()
corr_matrix
```

Out[151]:

	class_label	sensor0	sensor1	sensor2	sensor3	sensor4	sensor5	sensor6	sensor7	sensor8	sensor9
class_label	1.000000	0.543295	-0.252007	0.038775	0.433636	0.607623	0.224147	-0.007414	0.175612	0.660618	0.094907
sensor0	0.543295	1.000000	-0.215668	0.019521	0.253244	0.300760	0.164243	-0.007849	0.080306	0.361751	0.001317
sensor1	-0.252007	-0.215668	1.000000	0.056042	-0.111347	-0.130463	0.055401	-0.106082	0.031513	-0.163818	-0.023655
sensor2	0.038775	0.019521	0.056042	1.000000	0.093447	-0.023486	-0.008948	-0.057090	0.006899	-0.008308	-0.058528
sensor3	0.433636	0.253244	-0.111347	0.093447	1.000000	0.295136	0.038152	-0.105005	0.001243	0.287776	0.013732
sensor4	0.607623	0.300760	-0.130463	-0.023486	0.295136	1.000000	0.183916	-0.022690	0.090186	0.378306	0.023776
sensor5	0.224147	0.164243	0.055401	-0.008948	0.038152	0.183916	1.000000	-0.061238	0.020814	0.220014	0.071258
sensor6	-0.007414	-0.007849	-0.106082	-0.057090	-0.105005	-0.022690	-0.061238	1.000000	0.065211	-0.073309	0.040744
sensor7	0.175612	0.080306	0.031513	0.006899	0.001243	0.090186	0.020814	0.065211	1.000000	0.060028	0.059341
sensor8	0.660618	0.361751	-0.163818	-0.008308	0.287776	0.378306	0.220014	-0.073309	0.060028	1.000000	-0.033991
sensor9	0.094907	0.001317	-0.023655	-0.058528	0.013732	0.023776	0.071258	0.040744	0.059341	-0.033991	1.000000

Basically, this could be our solution to the task - just sort the features according to the absolute value of correlation coefficients:

In [152]:
```
result_1 = corr_matrix["class_label"].drop("class_label").apply(abs).sort_values(ascending=False)
result_1
```

Out[152]:
```
sensor8    0.660618
sensor4    0.607623
sensor0    0.543295
sensor3    0.433636
sensor1    0.252007
sensor5    0.224147
sensor7    0.175612
sensor9    0.094907
sensor2    0.038775
sensor6    0.007414
Name: class_label, dtype: float64
```

Write it to the file `result_1.csv`:

In [153]:
```
result_1.to_csv("result_1.csv", header=["score"], index_label="sensor")
```

However, *Pearson correlation* has several disadvantages:

1. Pearson correlation catches only linear dependency between variables, so if they are non-linearly correlated we would not notice that
2. It does not take into account the domain specific model constraints (for example importance of certain FNR or TPR, etc.)
3. It does not take into account model in which data is going to be used

Nevrtheless, it has some advantages too:

1. It is fast to compute which can be important in case of big datasets
2. It varies from $[-1, 1]$ which allows us to catch negative relationships

3. Decision tree features analysis

There are many statistical methods of correlation analysis but in this section we focus on *Decision Tree (or Forest)* based correlation as it is close to ML field and has natural heuristics for estimation of feature importance inside most of the algorithms (like Gini, information gain, etc.)

In [154]:
```
from sklearn.tree import DecisionTreeClassifier

feature_names = list(data.drop(["class_label"], axis=1).columns.values)
X = data[feature_names]
Y = data["class_label"]
rf = DecisionTreeClassifier(criterion="gini", max_features=None)
rf.fit(X, Y)
headers = ["name", "score"]
result_2 = pd.DataFrame(data=sorted(zip(X.columns, rf.feature_importances_), key=lambda x: x[1] * -1),
                        columns=["sensor", "score"])
result_2
```

Out[154]:

	sensor	score
0	sensor8	0.587708
1	sensor6	0.336496
2	sensor0	0.047977
3	sensor1	0.009679
4	sensor5	0.008333
5	sensor2	0.006667
6	sensor4	0.003140
7	sensor3	0.000000
8	sensor7	0.000000
9	sensor9	0.000000

As we can see, the importances are different from those in correlation matrix and some of them were not used at all.

Save the results in `result_2.csv`

In [155]:
```
result_2.to_csv("result_2.csv", index=False)
```

4. Model based features analysis

In *model based feature importance analysis*, we basically estimate the contribution of each feature into model prediction performance so that we learn the importance in exact model we want to use. *Permutation feature importance* works by randomly changing the values of each feature column, one column at a time, and then evaluating the input model. Important features are usually more sensitive to the shuffling process, and will thus result in higher importance scores. The advantage of this method is that it may capture domain and model specific dependencies but at the same time it may be very computationally expensive.

Assume that we want to use SVM as our production model. Let's estimate the importance of the features based on their contribution on this exact classifier:

In [156]:
```
from sklearn.model_selection import ShuffleSplit
from sklearn.svm import SVC
from sklearn.metrics import r2_score
from collections import defaultdict

clf = SVC()
scores = defaultdict(list)
X = np.array(X)
Y = np.array(Y)
for train_idx, test_idx in ShuffleSplit(100, .3).split(X):
    X_train = np.array([X[i] for i in train_idx])
    X_test = np.array([X[i] for i in test_idx])
    Y_train = np.array([Y[i] for i in train_idx])
    Y_test = np.array([Y[i] for i in test_idx])
    r = clf.fit(X_train, Y_train)
    acc = r2_score(Y_test, rf.predict(X_test))
    for i in range(X.shape[1]):
        X_t = X_test.copy()
        np.random.shuffle(X_t[:, i])
        shuff_acc = r2_score(Y_test, clf.predict(X_t))
        scores[feature_names[i]].append((acc-shuff_acc)/acc)

result_3 = pd.DataFrame(sorted([(np.mean(score), feat) for
              feat, score in scores.items()], reverse=True),
                        columns=["score", "sensor"]).reindex(columns=["sensor", "score"])
result_3
```

Out[156]:

	sensor	score
0	sensor8	0.790948
1	sensor4	0.649297
2	sensor0	0.508448
3	sensor3	0.395396
4	sensor7	0.339938
5	sensor1	0.330627
6	sensor9	0.322203
7	sensor6	0.318147
8	sensor2	0.303419
9	sensor5	0.301742

Note: in this table, score means how much accuracy we lose permutting respective feature.

Save the results in `result_3.csv` file:

In [157]:
```
result_3.to_csv("result_3.csv", index=False)
```

5. Conclusion

In this report we considered three types of feature importance analysis methods, each with its pros and cons described in respective sections. Most probably, the best choice is the last one (model based) as it selects features based on exact model we want to use. However, in current implementation, its scalability is limited by $O(n^2)$ complexity w.r.t. number of samples. So the final decision on which method should be made thoughtfully in each case as it depends heavily on dataset and domain.