Pr4_Titanic_sol

November 5, 2020

Dieses Notebook ist angelehnt an das *SciPy 2018 Scikit-learn Tutorial* von Andreas Mueller und Guillaume Lemaitre, verfügbar auf GitHub.

1 Praktikum: Session 4

Video

In dieser Session soll ein Modell entwickelt werden, welches basierend auf der Passagierliste der HMS Titanic das Überleben der einzelnen Passagiere vorhersagt.

Dazu verwenden wir eine Version der Titanic-Daten von hier. Solch externe Daten können auf unterschiedliche Arten bereitgestellt werden:

- direktes Einlesen aus der Quelle in einen Dateframe durch den Befehl `pd.read_excel('Adresse der Quelle')`.
- Herunterladen der Quelle und einlesen vom gewählten Speicherort (ebefalls durch `pd.read_excel`). Wenn Sie Colab verwenden können Sie durch das Menü links den Dateibaum einblenden und dorthin Dateien hochladen. Den genauen Pfad der Datei (um ihn in den `pd.read_***` Befehl einzufügen, erhalten Sie am einfachsten durch einen Rechtsklick auf die Datei, Kontextmenü "Pfad kopieren".)

Pandas kann viele Datenformate nativ einlesen, so gibt es z.B. auch `pd.read_csv()` etc.

Diese Daten werden nun eingelesen. Dabei werden die Daten der ersten Zeile als Spaltennamen interpretiert. Diese lassen wir direkt anzeigen:

[46]:
```
import os
import pandas as pd

titanic = pd.read_excel('http://biostat.mc.vanderbilt.edu/wiki/pub/Main/
 ↪DataSets/titanic3.xls')
#titanic = pd.read_csv(os.path.join('datasets', 'titanic3.csv'))
print(titanic.columns)
```

```
Index(['pclass', 'survived', 'name', 'sex', 'age', 'sibsp', 'parch', 'ticket',
       'fare', 'cabin', 'embarked', 'boat', 'body', 'home.dest'],
      dtype='object')
```

Was hat es mit diesen Spalten auf sich?

Hier eine knappe Beschreibung:

```
pclass          Passenger Class
                (1 = 1st; 2 = 2nd; 3 = 3rd)
survival        Survival
                (0 = No; 1 = Yes)
name            Name
sex             Sex
age             Age
sibsp           Number of Siblings/Spouses Aboard
parch           Number of Parents/Children Aboard
ticket          Ticket Number
fare            Passenger Fare
cabin           Cabin
embarked        Port of Embarkation
                (C = Cherbourg; Q = Queenstown; S = Southampton)
boat            Lifeboat
body            Body Identification Number
home.dest       Home/Destination
```

Welche dieser Features sollte man als kategorische Features auffassen, welche als numerische?

Kategorische Features: `name`, `sex`, `cabin`, `embarked`, `boat`, `body`, und `homedest`

Weitere kategorische Features: `pclass` - dieses ist zwar als Zahl angegeben, allerdings liegt hier keine algebraische Logik zugrunde.

Durch das Einlesen der Daten oben haben wir einen Pandas DataFrame angelegt. Verschaffen Sie sich ein erstes Gefühl für die Daten, indem Sie die ersten fünf Einträge betrachten.

[49]:
```
titanic.head()
```

[49]:

	pclass	survived	…	body	home.dest
0	1	1	…	NaN	St Louis, MO
1	1	1	…	NaN	Montreal, PQ / Chesterville, ON
2	1	0	…	NaN	Montreal, PQ / Chesterville, ON
3	1	0	…	135.0	Montreal, PQ / Chesterville, ON
4	1	0	…	NaN	Montreal, PQ / Chesterville, ON

[5 rows x 14 columns]

Ziel ist es, ein Modell zu bauen, welches für jeden Passagier vorhersagt, ob diese überlebt oder nicht. Welche Features sollte man für solch ein Modell sinnvollerweise verwenden? Was ist das gewünschte Label/Target?

- Legen Sie einen Vektor `labels` an, der die Labels enthält.
- Legen Sie entsprechend Ihrer Wahl eine Features-Matrix an.

Kontrollieren Sie, ob die erstellten Objekte sich so verhalten wie gewünscht, d.h. betrachten Sie diese kurz.

Hinweise: - Welche sind für die Fragestellung offensichtlich irrelevant, welche enthalten die gesuchte Information bereits auf offensichtliche Weise? - Einzelne Spalten können aus einem DataFrame `df`

extrahiert werden durch `df[['Spaltenname1', ..., 'SpaltennameN']]` - Ist man nur an den Werten interessiert, d.h. an einem Vektor ohne die zugehörigen Indizierung, so kann man noch `.values` anhängen.

Wir verwenden die Features "pclass", "sex", "age", "sibsp", "parch", "fare" und "embarked".

[50]:
```
labels = titanic['survived'].values
features = titanic[['pclass', 'sex', 'age', 'sibsp', 'parch', 'fare',␣
 ↪'embarked']]
```

[52]:
```
features.shape
```

[52]:
```
(1309, 7)
```

[53]:
```
features.head()
```

[53]:

	pclass	sex	age	sibsp	parch	fare	embarked
0	1	female	29.0000	0	0	211.3375	S
1	1	male	0.9167	1	2	151.5500	S
2	1	female	2.0000	1	2	151.5500	S
3	1	male	30.0000	1	2	151.5500	S
4	1	female	25.0000	1	2	151.5500	S

Der DataFrame `features` enthält nun nur noch nützliche Features, allerdings liegen diese noch nicht in einer nutzbaren Form vor (nicht-numerische Daten). Im Folgenden sollen die Daten aufbereitet werden.

1.1 Datenaufbereitung mit Pandas

Da unsere Daten in einem DataFrame vorliegen, können wir Möglichkeiten der Bibliothek `Pandas` (in welcher DataFrames definiert sind) nutzen, um die nötigen Manipulationen durchzuführen.

Offensichtlich verändert werden müssen die Features `sex` und `embarked`; die naheliegende Option ist ein One-Hot-Encoding. Eine solche One-Hot-Codierung aller nicht-numerischen Spalten kann in Pandas sehr einfach z.B. durch die Methode `get_dummies()` durchgeführt werden. Wenden Sie diese Methode auf die Daten an und betrachten Sie das Ergebnis.

Hinweis: Die Methode stammt aus der Bibliothek `pandas`, welche wir ganz oben importiert haben; wir haben sie unter dem (Standard-)Kürzel `pd` bereitgestellt. Somit können Sie die gewünschete Methode aufrufen mit `pd.get_dummies(zu_verarbeitende_Daten)` aufrufen.

[54]:
```
pd.get_dummies(features).head()
```

[54]:

	pclass	age	sibsp	parch	…	sex_male	embarked_C	embarked_Q	embarked_S
0	1	29.0000	0	0	…	0	0	0	1
1	1	0.9167	1	2	…	1	0	0	1
2	1	2.0000	1	2	…	0	0	0	

```
1
3       1  30.0000      1      2  …         1           0           0
1
4       1  25.0000      1      2  …         0           0           0
1

[5 rows x 10 columns]
```

Oben hatten wir überlegt, dass es sinnvoll wäre, auch das Feature `pclass` als kategorisch aufzufassen. Verwenden Sie wieder die `get_dummies()` Methode, um auch `pclass` durch One-Hot-Encoding darzustellen.

Hinweis: `get_dummies()` kann das optionale Argument `columns` übergeben werden. Durch dieses kann eine Liste an Features explizit angegeben werden, die One-Hot kodiert werden sollen.

[55]:
```
features_dummies = pd.get_dummies(features, columns=['pclass', 'sex',␣
↪'embarked'])
features_dummies.head()
```

[55]:
```
       age  sibsp  parch  …  embarked_C  embarked_Q  embarked_S
0  29.0000      0      0  …           0           0           1
1   0.9167      1      2  …           0           0           1
2   2.0000      1      2  …           0           0           1
3  30.0000      1      2  …           0           0           1
4  25.0000      1      2  …           0           0           1

[5 rows x 12 columns]
```

Damit liegen die Daten nun in einer geeigneten Form vor. Den DataFrame brauchen wir für unser Modell eigentlich nicht mehr, daher extrahieren wir die Daten daraus mit `data = features_dummies.values`.

[56]:
```
data = features_dummies.values
```

[59]:
```
data
```

[59]:
```
array([[29.    ,  0.    ,  0.    , …,  0.    ,  0.    ,  1.    ],
       [ 0.9167,  1.    ,  2.    , …,  0.    ,  0.    ,  1.    ],
       [ 2.    ,  1.    ,  2.    , …,  0.    ,  0.    ,  1.    ],
       …,
       [26.5   ,  0.    ,  0.    , …,  1.    ,  0.    ,  0.    ],
       [27.    ,  0.    ,  0.    , …,  1.    ,  0.    ,  0.    ],
       [29.    ,  0.    ,  0.    , …,  0.    ,  0.    ,  1.    ]])
```

1.2 Fehlende Daten

Wir müssen aber noch prüfen, ob evtl. Daten fehlen, d.h. ob manche Einträge den Wert `NaN` (not a number) haben und damit ungültig sind. Das kann z.B. mit Hilfe der Methode `isnan()` gemacht

werden. Diese steht in der Bibliothek `numpy` bereit (welche standardmäßig mit dem `np` importiert wird) und liefert bei Anwendung auf ein Array ein Array der gleichen Form zurück, welches nur die Einträge `True` und `False` hat. `True` steht an jeder Position, an der das Ausgangsarray den Wert `NaN` hat, `False` überall dort, wo gültige Daten vorliegen.

Somit interessiert uns eigentlich nur, ob dieses große "`True/False`-Array" irgendwo den Wert `True` hat. Dies kann durch die Methode `any()` realisiert werden.

[60]:
```
import numpy as np
np.isnan(data)
```

[60]:
```
array([[False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       …,
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False],
       [False, False, False, …, False, False, False]])
```

[61]:
```
np.isnan(data).any()
```

[61]:
```
True
```

Da es also fehlende Daten gibt, müssen diese ersetzt werden. Dafür wollen wir den `SimpleImputer` verwenden.

VORSICHT: Bevor ein Imputer verwendet wird, müssen die Daten in Trainings- und Testdaten aufgeteilt werden! Warum?

Der Imputer verwendet die gesamten ihm "gefütterten" Daten, um zu bestimmen, was mit den fehlenden Daten gemacht wird (z.B. durch den Mittelwert ersetzen). Somit geht Information aus der Datenmenge, auf die der Imputer Zugriff hat überall dort ein, wo Daten fehlen. Damit werden möglicherweise die Trainingsdaten verunreinigt.

Konkret: In den Trainingsdaten fehlen Werte für den Fahrpreis. Wenn der Imputer nun den Mittelwert aus den *gesamten* Daten bildet, um fehlende zu ersetzen, geht Information aus dem Testset implizit in das Training ein.

- Teilen Sie die vorliegenden (und bereits aufbereiteten) Daten auf in Trainings- und Testdaten.
- Verwenden Sie den `SimpleImputer` um fehlende Werte zu ergänzen.
- Prüfen Sie nochmals, ob nun noch `NaN` vorhanden ist.

[62]:
```
from sklearn.model_selection import train_test_split
from sklearn.impute import SimpleImputer

train_data, test_data, train_labels, test_labels = train_test_split(data,␣
 ↪labels, random_state=0)

imp = SimpleImputer()
imp.fit(train_data)
```

```
train_data_finite = imp.transform(train_data)
test_data_finite = imp.transform(test_data)
```

[63]:
```
print("Fehlernde Werte in Trainingsdaten: ", np.isnan(train_data_finite).any())
print("Fehlernde Werte in Testdaten: ", np.isnan(test_data_finite).any())
```

```
Fehlernde Werte in Trainingsdaten:  False
Fehlernde Werte in Testdaten:  False
```

1.3 Betrachtung von Modellen

Nun sind die Daten soweit aufbereitet, dass man ein Modell damit trainieren kann. Es stellt sich wie immer die Frage: Welches Ergebnis ist gut?

1.3.1 Baseline: Dummy Classifier

Verwenden Sie den `DummyClassifier` mit der Strategie `most_frequent`, um eine untere Grenze für eine akzeptable Vorhersagegenauigkeit zu erhalten. Was macht dieser Dummy Classifier?

Hinweis: Diesen Classifier finden Sie in `sklearn.dummy`.

[64]:
```
from sklearn.dummy import DummyClassifier

clf = DummyClassifier('most_frequent')
clf.fit(train_data_finite, train_labels)
print("Genauigkeit Dummy Classifier: %f"
      % clf.score(test_data_finite, test_labels))
```

```
Genauigkeit Dummy Classifier: 0.634146
```

1.3.2 Naive Bayes

Verwenden Sie einen Naive Bayes Classifier und bestimmen Sie dessen Genauigkeit.

[65]:
```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()
gnb.fit(train_data_finite, train_labels)
print("Genauigkeit Gaussian Naive Bayes: %f"
      % gnb.score(test_data_finite, test_labels))
```

```
Genauigkeit Gaussian Naive Bayes: 0.780488
```

1.3.3 Ensemble Methode: RandomForestClassifier

Verwenden Sie den RandomForestClassifier und bestimmen Sie dessen Genauigkeit. *Hinweis:* Diesen finden Sie in `sklearn.ensemble`.

[70]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(random_state=0).fit(train_data_finite, train_labels)
```

```
print("Genauigkeit Random Forest: %f" % rf.score(test_data_finite, test_labels))
```

```
Genauigkeit Random Forest: 0.777439
```

Entfernen Sie die Features `embark` und `parch` aus dem Datensatz und trainieren und bewerten Sie erneut das gleiche RandomForest Modell.

[74]:
```
features_dummies_sub = pd.get_dummies(features[['pclass', 'sex', 'age',␣
 ↪'sibsp',  'fare']])
data_sub = features_dummies_sub.values

train_data_sub, test_data_sub, train_labels, test_labels =␣
 ↪train_test_split(data_sub, labels, random_state=0)

imp = SimpleImputer()
imp.fit(train_data_sub)
train_data_finite_sub = imp.transform(train_data_sub)
test_data_finite_sub = imp.transform(test_data_sub)

rf = RandomForestClassifier(random_state=0).fit(train_data_finite_sub,␣
 ↪train_labels)
print("Genauigkeit Random Forest ohne 'embark' und 'parch': %f" % rf.
 ↪score(test_data_finite_sub, test_labels))
```

```
Genauigkeit Random Forest ohne 'embark' und 'parch': 0.798780
```

1.3.4 Support Vector Classifier (SVC)

Verwenden Sie einen Support Vector Classifier und bestimmen Sie dessen Genauigkeit.

[75]:
```
from sklearn.svm import SVC

svc = SVC()
svc.fit(train_data_finite, train_labels)
print("Genauigkeit SVC: %f"
      % svc.score(test_data_finite, test_labels))
```

```
Genauigkeit SVC: 0.689024
```

Der Classifier SVC biete im Gegensatz zu Naive Bayes einige Hyperparameter, die es zu wählen gilt. Verwenden Sie eine GridSearch, um eine geeignete Kombination zu finden.

[76]:
```
from sklearn.model_selection import GridSearchCV

param_grid = {'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf'], 'gamma': [0.01, 0.
 ↪1, 1]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=3, verbose=3, n_jobs=-1)
```

[ ]:
```
grid.fit(train_data_finite, train_labels)
```

```
Fitting 3 folds for each of 18 candidates, totalling 54 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 16 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 out of  54 | elapsed:   11.3s remaining:    3.2s
[Parallel(n_jobs=-1)]: Done  54 out of  54 | elapsed:   49.4s finished
```

[ ]:
```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'C': [0.1, 1, 10], 'kernel': ['linear', 'rbf'], 'gamma':
 [0.01, 0.1, 1]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
       scoring=None, verbose=3)
```

[ ]:
```
print("Genauigkeit bester betrachteter SVC: %f"
      % grid.score(test_data_finite, test_labels))
```

```
Genauigkeit bester betrachteter SVC: 0.777439
```

[ ]:
```
grid.best_params_
```

[ ]:
```
{'C': 0.1, 'gamma': 0.01, 'kernel': 'linear'}
```

Wir wollen das Grid um die besten gefundenen Hyperparameterwerte feiner machen in der Hoffnung, eine weitere Verbesserung zu erzielen:

[ ]:
```
param_grid = {'C': [0.01, 0.05, 0.1, 0.2], 'kernel': ['linear'], 'gamma': [0.
 ↪001, 0.005, 0.01, 0.05]}
grid = GridSearchCV(SVC(), param_grid=param_grid, cv=3, verbose=3, n_jobs=-1)
grid.fit(train_data_finite, train_labels)
```

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 16 concurrent workers.
[Parallel(n_jobs=-1)]: Done  17 out of  48 | elapsed:    0.0s remaining:    0.1s
[Parallel(n_jobs=-1)]: Done  48 out of  48 | elapsed:   13.2s finished
```

[ ]:
```
GridSearchCV(cv=3, error_score='raise-deprecating',
       estimator=SVC(C=1.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto_deprecated',
  kernel='rbf', max_iter=-1, probability=False, random_state=None,
  shrinking=True, tol=0.001, verbose=False),
       fit_params=None, iid='warn', n_jobs=-1,
       param_grid={'C': [0.01, 0.05, 0.1, 0.2], 'kernel': ['linear'], 'gamma':
 [0.001, 0.005, 0.01, 0.05]},
       pre_dispatch='2*n_jobs', refit=True, return_train_score='warn',
```

```
        scoring=None, verbose=3)
```

[ ]:
```
print("Genauigkeit bester betrachteter SVC: %f"
      % grid.score(test_data_finite, test_labels))
```

```
Genauigkeit bester betrachteter SVC: 0.777439
```

[ ]:
```
grid.best_params_
```

[ ]:
```
{'C': 0.05, 'gamma': 0.001, 'kernel': 'linear'}
```

Das hatte nicht den gewünschten Effekt...

[ ]: