Logistic Regression in scikit-learn

Introduction

Generally, the process for fitting a logistic regression model using scikit-learn is very similar to that which you previously saw for `statsmodels`. One important exception is that scikit-learn will not display statistical measures such as the p-values associated with the various features. This is a shortcoming of scikit-learn, although scikit-learn has other useful tools for tuning models which we will investigate in future lessons.

The other main process of model building and evaluation which we didn't discuss previously is performing a train-test split. As we saw in linear regression, model validation is an essential part of model building as it helps determine how our model will generalize to future unseen cases. After all, the point of any model is to provide future predictions where we don't already know the answer but have other informative data (`X`).

With that, let's take a look at implementing logistic regression in scikit-learn using dummy variables and a proper train-test split.

Objectives

You will be able to:

- Fit a logistic regression model using scikit-learn

Importing the Data

In [1]:
```
import pandas as pd

df = pd.read_csv('titanic.csv')
df.head()
```

Out[1]:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Defining `X` and `y`

To start out, we'll consider `y` to be the target variable (`Survived`) and everything else to be `X` .

In [2]:
```
y = df["Survived"]
X = df.drop("Survived", axis=1)
```

Train-Test Split

Specifying a `random_state` means that we will get consistent results even if the kernel is restarted.

In [3]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Preprocessing

Dealing with Missing Data

Some of the data is missing, which won't work with a scikit-learn model:

In [4]:
```
X_train.isna().sum()
```

Out[4]:
```
PassengerId      0
Pclass           0
Name             0
Sex              0
Age            133
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin          511
Embarked         2
dtype: int64
```

For `Cabin` and `Embarked` (categorical features), we'll manually fill this in with "missing" labels:

In [5]:
```
X_train_fill_na = X_train.copy()
X_train_fill_na.fillna({"Cabin":"cabin_missing", "Embarked":"embarked_missing"}, inplace=True)
X_train_fill_na.isna().sum()
```

Out[5]:
```
PassengerId      0
Pclass           0
Name             0
Sex              0
Age            133
SibSp            0
Parch            0
Ticket           0
Fare             0
Cabin            0
Embarked         0
dtype: int64
```

For `Age` (a numeric feature), we'll use a `SimpleImputer` from scikit-learn (documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.impute.SimpleImputer.html)) to fill in the mean:

In [6]:
```
from sklearn.impute import SimpleImputer

imputer = SimpleImputer()

imputer.fit(X_train_fill_na[["Age"]])
age_imputed = pd.DataFrame(
    imputer.transform(X_train_fill_na[["Age"]]),
    # index is important to ensure we can concatenate with other columns
    index=X_train_fill_na.index,
    columns=["Age"]
)

X_train_fill_na["Age"] = age_imputed
X_train_fill_na.isna().sum()
```

Out[6]:
```
PassengerId    0
Pclass         0
Name           0
Sex            0
Age            0
SibSp          0
Parch          0
Ticket         0
Fare           0
Cabin          0
Embarked       0
dtype: int64
```

Dealing with Categorical Data

Some of the columns of `X_train_fill_na` currently contain categorical data (i.e. Dtype `object`):

In [7]:
```
X_train_fill_na.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 668 entries, 105 to 684
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  668 non-null    int64  
 1   Pclass       668 non-null    int64  
 2   Name         668 non-null    object 
 3   Sex          668 non-null    object 
 4   Age          668 non-null    float64
 5   SibSp        668 non-null    int64  
 6   Parch        668 non-null    int64  
 7   Ticket       668 non-null    object 
 8   Fare         668 non-null    float64
 9   Cabin        668 non-null    object 
 10  Embarked     668 non-null    object 
dtypes: float64(2), int64(4), object(5)
memory usage: 62.6+ KB
```

In [8]:
```
X_train_categorical = X_train_fill_na.select_dtypes(exclude=["int64", "float64"]).copy()
X_train_categorical
```

Out[8]:

	Name	Sex	Ticket	Cabin	Embarked
105	Mionoff, Mr. Stoytcho	male	349207	cabin_missing	S
68	Andersson, Miss. Erna Alexandra	female	3101281	cabin_missing	S
253	Lobb, Mr. William Arthur	male	A/5. 3336	cabin_missing	S
320	Dennis, Mr. Samuel	male	A/5 21172	cabin_missing	S
706	Kelly, Mrs. Florence "Fannie"	female	223596	cabin_missing	S
...	...	...	...	...	...
835	Compton, Miss. Sara Rebecca	female	PC 17756	E49	C
192	Andersen-Jensen, Miss. Carla Christine Nielsine	female	350046	cabin_missing	S
629	O'Connell, Mr. Patrick D	male	334912	cabin_missing	Q
559	de Messemaeker, Mrs. Guillaume Joseph (Emma)	female	345572	cabin_missing	S
684	Brown, Mr. Thomas William Solomon	male	29750	cabin_missing	S

668 rows × 5 columns

`OneHotEncoder` from scikit-learn ([documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html)) can be used to convert categorical variables into dummy one-hot encoded variables:

In [9]:
```
from sklearn.preprocessing import OneHotEncoder
import numpy as np

ohe = OneHotEncoder(handle_unknown="ignore", sparse=False)

ohe.fit(X_train_categorical)
X_train_ohe = pd.DataFrame(
    ohe.transform(X_train_categorical),
    # index is important to ensure we can concatenate with other columns
    index=X_train_categorical.index,
    # we are dummying multiple columns at once, so stack the names
    columns=np.hstack(ohe.categories_)
)
X_train_ohe
```

Out[9]:

	Abbing, Mr. Anthony	Abbott, Mr. Rossmore Edward	Abelson, Mrs. Samuel (Hannah Wizosky)	Adahl, Mr. Mauritz Nils Martin	Adams, Mr. John	Aks, Mrs. Sam (Leah Rosen)	Albimona, Mr. Nassef Cassem	Alexander, Mr. William	Alhomaki, Mr. Ilmari Rudolf	Allen, Miss. Elisabeth Walton	...	F33	F38	F4	G6	T	cabin_missing	C	Q
105	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	0.0	1.0	0.0	0.0
68	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	0.0	1.0	0.0	0.0
253	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	0.0	1.0	0.0	0.0
320	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	0.0	1.0	0.0	0.0
706	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	0.0	1.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
835	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	0.0	0.0	1.0	0.0
192	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	0.0	1.0	0.0	0.0
629	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	0.0	1.0	0.0	1.0
559	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	0.0	1.0	0.0	0.0
684	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	0.0	1.0	0.0	0.0

668 rows × 1336 columns

Wow! That's a lot of columns! Way more than is useful in practice: we now have columns for each of the passenger's names. This is an example of what not to do. Let's try that again, this time being mindful of which variables we actually want to include in our model.

Instead of just selecting every single categorical feature for dummying, let's only include the ones that make sense as categories rather than being the names of individual people:

In [10]:
```
categorical_features = ["Sex", "Cabin", "Embarked"]
X_train_categorical = X_train_fill_na[categorical_features].copy()
X_train_categorical
```

Out[10]:

	Sex	Cabin	Embarked
105	male	cabin_missing	S
68	female	cabin_missing	S
253	male	cabin_missing	S
320	male	cabin_missing	S
706	female	cabin_missing	S
...	...	...	...
835	female	E49	C
192	female	cabin_missing	S
629	male	cabin_missing	Q
559	female	cabin_missing	S
684	male	cabin_missing	S

668 rows × 3 columns

In [11]:
```
ohe.fit(X_train_categorical)

X_train_ohe = pd.DataFrame(
    ohe.transform(X_train_categorical),
    index=X_train_categorical.index,
    columns=np.hstack(ohe.categories_)
)
X_train_ohe
```

Out[11]:

	female	male	A10	A14	A16	A19	A20	A23	A24	A31	...	F33	F38	F4	G6	T	cabin_missing	C	Q	S	embarked_missing
105	0.0	1.0	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	1.0	0.0	0.0	1.0	0.0
68	1.0	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	0.0	1.0	0.0	0.0	1.0	0.0
253	0.0	1.0	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	1.0	0.0	0.0	1.0	0.0
320	0.0	1.0	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	1.0	0.0	0.0	1.0	0.0
706	1.0	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	0.0	1.0	0.0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
835	1.0	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	0.0	0.0	1.0	0.0	0.0	0.0
192	1.0	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	0.0	1.0	0.0	0.0	1.0	0.0
629	0.0	1.0	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	1.0	0.0	1.0	0.0	0.0
559	1.0	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	0.0	1.0	0.0	0.0	1.0	0.0
684	0.0	1.0	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	1.0	0.0	0.0	1.0	0.0

668 rows × 130 columns

That's still a lot of columns, but we no longer have more columns than records!

Normalization

Now let's look at the numeric features. This time we'll also pay more attention to the meaning of the features, and only include relevant ones (e.g. not including `PassengerId` because this is a data artifact, not a true feature).

Another important data preparation practice is to normalize your data. That is, if the features are on different scales, some features may impact the model more heavily then others. To level the playing field, we often normalize all features to a consistent scale of 0 to 1.

As you can see, our features are currently not on a consistent scale:

In [12]:
```
numeric_features = ["Pclass", "Age", "SibSp", "Fare"]
X_train_numeric = X_train_fill_na[numeric_features].copy()
X_train_numeric
```

Out[12]:

	Pclass	Age	SibSp	Fare
105	3	28.0	0	7.8958
68	3	17.0	4	7.9250
253	3	30.0	1	16.1000
320	3	22.0	0	7.2500
706	2	45.0	0	13.5000
...	...	...	...	...
835	1	39.0	1	83.1583
192	3	19.0	1	7.8542
629	3	29.9	0	7.7333
559	3	36.0	1	17.4000
684	2	60.0	1	39.0000

668 rows × 4 columns

Let's use a `MinMaxScaler` from scikit-learn ([documentation here (https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html)](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html)) with default parameters to create a maximum value of 1 and a minimum value of 0. This will work well with our binary one-hot encoded data.

In [13]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaler.fit(X_train_numeric)
X_train_scaled = pd.DataFrame(
    scaler.transform(X_train_numeric),
    # index is important to ensure we can concatenate with other columns
    index=X_train_numeric.index,
    columns=X_train_numeric.columns
)
X_train_scaled
```

Out[13]:

	Pclass	Age	SibSp	Fare
105	1.0	0.344510	0.000	0.015412
68	1.0	0.205849	0.500	0.015469
253	1.0	0.369721	0.125	0.031425
320	1.0	0.268877	0.000	0.014151
706	0.5	0.558805	0.000	0.026350
...	...	...	...	...
835	0.0	0.483172	0.125	0.162314
192	1.0	0.231060	0.125	0.015330
629	1.0	0.368461	0.000	0.015094
559	1.0	0.445355	0.125	0.033963
684	0.5	0.747889	0.125	0.076123

668 rows × 4 columns

Then we concatenate everything together:

In [14]:
```
X_train_full = pd.concat([X_train_scaled, X_train_ohe], axis=1)
X_train_full
```

Out[14]:

	Pclass	Age	SibSp	Fare	female	male	A10	A14	A16	A19	...	F33	F38	F4	G6	T	cabin_missing	C	Q	S	embarked_missing
105	1.0	0.344510	0.000	0.015412	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
68	1.0	0.205849	0.500	0.015469	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
253	1.0	0.369721	0.125	0.031425	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
320	1.0	0.268877	0.000	0.014151	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
706	0.5	0.558805	0.000	0.026350	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
835	0.0	0.483172	0.125	0.162314	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
192	1.0	0.231060	0.125	0.015330	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
629	1.0	0.368461	0.000	0.015094	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
559	1.0	0.445355	0.125	0.033963	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
684	0.5	0.747889	0.125	0.076123	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0

668 rows × 134 columns

Fitting a Model

Now let's fit a model to the preprocessed training set. In scikit-learn, you do this by first creating an instance of the `LogisticRegression` class. From there, then use the `.fit()` method from your class instance to fit a model to the training data.

In [15]:
```
from sklearn.linear_model import LogisticRegression

logreg = LogisticRegression(fit_intercept=False, C=1e12, solver='liblinear')
model_log = logreg.fit(X_train_full, y_train)
model_log
```

Out[15]:
```
LogisticRegression(C=1000000000000.0, fit_intercept=False, solver='liblinear')
```

Model Evaluation

Now that we have a model, lets take a look at how it performs.

Performance on Training Data

First, how does it perform on the training data?

In the cell below, `0` means the prediction and the actual value matched, whereas `1` means the prediction and the actual value did not match.

In [16]:
```
y_hat_train = logreg.predict(X_train_full)

train_residuals = np.abs(y_train - y_hat_train)
print(pd.Series(train_residuals, name="Residuals (counts)").value_counts())
print()
print(pd.Series(train_residuals, name="Residuals (proportions)").value_counts(normalize=True))
```

```
0    567
1    101
Name: Residuals (counts), dtype: int64

0    0.848802
1    0.151198
Name: Residuals (proportions), dtype: float64
```

Not bad; our classifier was about 85% correct on our training data!

Performance on Test Data

Now let's apply the same preprocessing process to our test data, so we can evaluate the model's performance on unseen data.

In [17]:
```
# Filling in missing categorical data
X_test_fill_na = X_test.copy()
X_test_fill_na.fillna({"Cabin":"cabin_missing", "Embarked":"embarked_missing"}, inplace=True)

# Filling in missing numeric data
test_age_imputed = pd.DataFrame(
    imputer.transform(X_test_fill_na[["Age"]]),
    index=X_test_fill_na.index,
    columns=["Age"]
)
X_test_fill_na["Age"] = test_age_imputed

# Handling categorical data
X_test_categorical = X_test_fill_na[categorical_features].copy()
X_test_ohe = pd.DataFrame(
    ohe.transform(X_test_categorical),
    index=X_test_categorical.index,
    columns=np.hstack(ohe.categories_)
)

# Normalization
X_test_numeric = X_test_fill_na[numeric_features].copy()
X_test_scaled = pd.DataFrame(
    scaler.transform(X_test_numeric),
    index=X_test_numeric.index,
    columns=X_test_numeric.columns
)

# Concatenating categorical and numeric data
X_test_full = pd.concat([X_test_scaled, X_test_ohe], axis=1)
X_test_full
```

Out[17]:

	Pclass	Age	SibSp	Fare	female	male	A10	A14	A16	A19	...	F33	F38	F4	G6	T	cabin_missing	C	Q	S	embarked_missing
495	1.0	0.368461	0.000	0.028221	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
648	1.0	0.368461	0.000	0.014737	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
278	1.0	0.079793	0.500	0.056848	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	1.0	0.0	0.0
31	0.0	0.368461	0.125	0.285990	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
255	1.0	0.357116	0.000	0.029758	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
167	1.0	0.558805	0.125	0.054457	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
306	0.0	0.368461	0.000	0.216430	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0
379	1.0	0.231060	0.000	0.015176	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0
742	0.0	0.256271	0.250	0.512122	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
10	1.0	0.041977	0.125	0.032596	1.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0

223 rows × 134 columns

In [18]:
```
y_hat_test = logreg.predict(X_test_full)

test_residuals = np.abs(y_test - y_hat_test)
print(pd.Series(test_residuals, name="Residuals (counts)").value_counts())
print()
print(pd.Series(test_residuals, name="Residuals (proportions)").value_counts(normalize=True))
```

```
0    175
1     48
Name: Residuals (counts), dtype: int64

0    0.784753
1    0.215247
Name: Residuals (proportions), dtype: float64
```

And still about 78% accurate on our test data!

Summary

In this lesson, you took a more complete look at a data science pipeline for logistic regression, splitting the data into training and test sets and using the model to make predictions. You'll practice this on your own in the upcoming lab before having a more detailed discussion of more nuanced methods for evaluating a classifier's performance.