**Fourth Phase Evaluation Project  Pre-processing Pipeline**

**Contents**

[**1. Introduction**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#1.-Introduction)

[**2. Scaling and transforming numerical data**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#2.-Scaling-and-transforming-numerical-data)

[**3. Imputation**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#3.-Imputation)

[**4. Encoding categorical data**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#4.-Encoding-categorical-data)

[**5. ColumnTransformer - applying different transformations to different columns**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#5.-ColumnTransformer---applying-different-transformations-to-different-columns)

[**6. Replacing a column with a custom function**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#6.-Replacing-a-column-with-a-custom-function)

[**7. Feature engineering - adding a column**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#7.-Feature-engineering---adding-a-column)

[**8. Drop columns**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#8.-Drop-columns)

[**9. Named transformer output**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#9.-Named-transformer-output)

[**10. Modelling using the pipeline**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#10.-Modelling-using-the-pipeline)

[**11. Testing preprocessing choices using GridSearchCV**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#11.-Testing-preprocessing-choices-using-GridSearchCV)

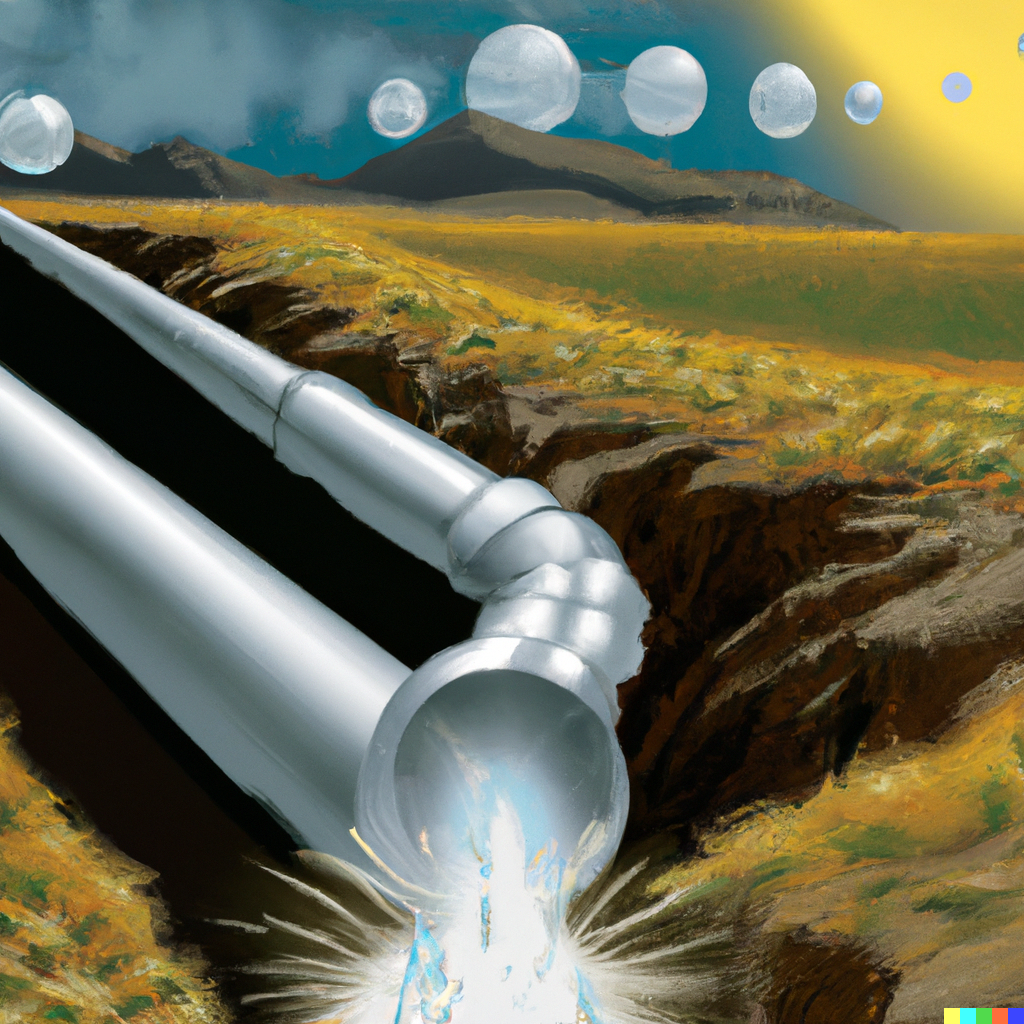
[**12. Limitations of pipelines**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#12.-Limitations-of-pipelines)

[**13. Conclusions**](https://www.kaggle.com/code/nnjjpp/pipelines-for-preprocessing-a-tutorial#13.-Conclusions)

from IPython.display import Image

Image("../input/pipeline-diagrams/pipeline.png",width=1200)

Out[1]:



**1. Introduction**

Pipelines are a powerful tool for organizing your machine-learning workflow. In this notebook I show how you can use scikit-learn pipelines to set up your data preprocessing in a clean and efficient way, eliminating duplicated code, minimising the chance of errors, and allowing tuning of preprocessing steps to create better predictions.

Some common preprocessing steps that we might want to include in a pipeline are:

1. Scaling and transforming numerical data
2. Encoding a categorical column
3. Imputing missing values
4. Processing and replacing a column
5. Adding columns derived from one or more columns (e.g. indicator and aggregation columns)
6. Dropping columns

By setting these processing steps as scikit-learn transformers (rather than processing the data with plain functions), we can apply these data transformations more efficiently.

1.0 Incorrect/inefficient approaches

one of two workflows, firstly appending the test data to the training data and applying preprocessing to the combined dataset. A simple example, is encoding a categorical variable in both training and test data:

both = pd.concat([train.drop(target\_variable, axis=1), test])

both[categorical\_variable] = OrdinalEncoder.fit\_transform(both[categorical\_variable])

train = both.iloc[train.shape[0]:,:]

test = both.iloc[:train.shape[0],:]

The main reason this is a bad idea is because of data leakage. Basically we are building a substandard model by applying preprocessing to training and test data simultaneously as information from the test data can leak out into the preprocessed training data. (See section 1.2.)

Another common approach is simply applying the preprocessing twice:

oe = OrdinalEncoder()

train[categorical\_variable] = oe.fit\_transform(train[categorical\_variable])

test[categorical\_variable] = oe.transform(test[categorical\_variable])

Done properly (i.e. calling transform rather than fit\_transform on the test data), this eliminates the data leakage problem.

However, if we're using a cross-validation scheme, this should be applied the training and test data *within each CV fold*. Thus, this approach can suffer from issues with repeated code, as we'd need to apply the transform multiple times within the workbook. Moreover, if there is more than one pre-processing step, combining all the steps into a pipeline means that we only need to perform the fit and transform (or predict) methods on the pipeline rather than each individual step.

1.1 What is a pipeline?

A pipeline is a series of processing and/or modelling steps bundled together into one object.

Pipeline can be used to chain multiple estimators into one.

Here, an 'estimator' typically means a data transformer (e.g. StandardScaler, or OneHotEncoder), or a predictive model (such as LinearRegression). The most common use case is chaining a number of data preprocessing steps together before a model. Typically, we have a number of transformers that are fitted and then applied to the data, and a predictive model at the end of the pipeline chain.

Intermediate steps of the pipeline must be ‘transforms’, that is, they must implement fit and transform methods. The final estimator only needs to implement fit.

Often the final estimator has predict and score, etc. methods as well. Alternatively we could have a chain of transformations without a model at the end, and we'd typically call transform or fit\_transform on the pipeline.

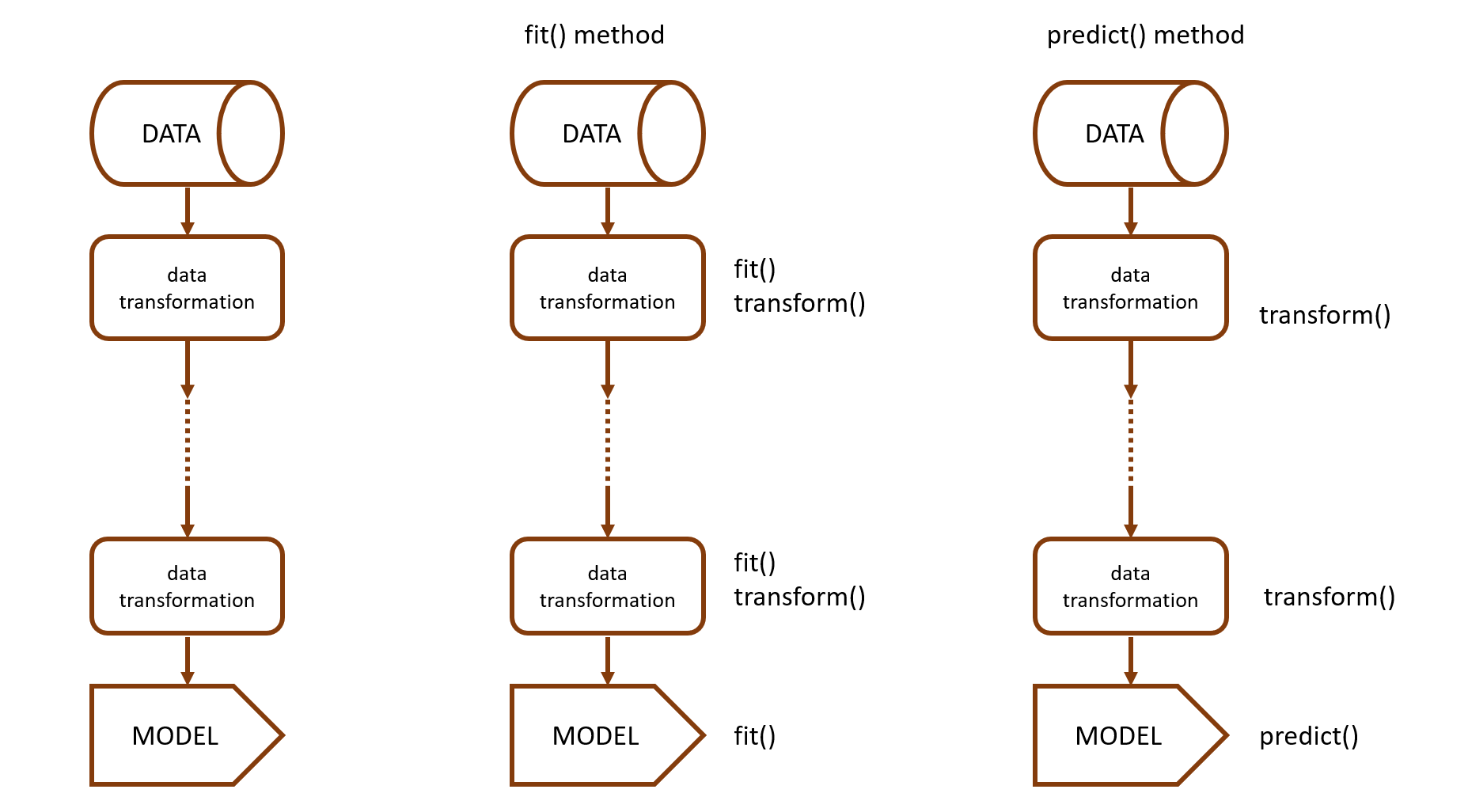
The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters.

Quite often we have a series of transformations (e.g. encoding categorical variables, power transforms for numerical variables, feature engineering, imputing missing values) that we'd like to apply to a dataset, and a candidate model (or models). Notice that preprocessing steps in scikit-learn all use fit() and transform() methods, and the machine learning estimators and models have similarly structured APIs, namely the fit() and predict() methods. The idea behind a pipeline is to use these similar APIs to link together transformations and models so that the output of one step feeds into the input of the next step in the pipeline.

Note that when fit is called on the pipeline, each step is fitted and transformed, whereas when predict (or score, transform, etc.) is called on the entire pipeline, each estimator is only transformed, not fitted.

unfold\_moreShow hidden code

Out[2]:



 Why pipelines?

There are several benefits of using a pipeline, and these include:

1. Eliminating duplicated code.

An important tenet of software engineering is ["Don't repeat yourself"](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself), or DRY. Copying and pasting code used to process training data to process the testing data is fraught with problems. Changes made to the data pre-processing workflow can be easily missed or errors made if you need to change the code in two (or more) places. Using a pipeline eliminates duplicated code by combining preprocessing steps into one class which can be easily instantiated again.

1. Avoiding [data](https://machinelearningmastery.com/data-preparation-without-data-leakage/) [leakage](https://jfrog.com/community/data-science/be-careful-from-data-leakage/) by processing training and test data separately.

One (substandard) way to get around duplicated code is to combine the training and test data and apply the preprocessing to the combined dataset. I used to do this, and I see it quite commonly on, but it is [not recommended](https://community.alteryx.com/t5/Data-Science/Dealing-with-Data-Leakage/ba-p/827583). Briefly, preprocessing training and test data concurrently means test data information (e.g. distributions of features) is seen by the fit() method, and this will provide an overly optimistic test score, which ultimately leads to degraded performance on new predictions.

In contrast, we see from the above diagram that when the pipeline's predict (or score, transform, etc.) method is called, the pipeline is not re-fitted on the test dataset, but the pipeline is just transformed without fitting. Thus we avoid data leakage from preprocessing using a pipeline in this way.

1. Embedding preprocessing in a cross-validation scheme.

We saw in section 1.0 that data leakage from preprocessing can be avoided by calling fit\_transform on the training data and transform on the test data. However, if we're using a cross-validation scheme, the preprocessing should be applied in each fold, first being fitted to the CV training data and then applied (via transform) to the CV test data. This can be accomplished with model validation schemes like cross\_validate or GridSearchCV by passing the pipeline in as the estimator, rather than the final model, and the raw data as 'X' and 'y'.

1. Code readability.

By eliminating duplicated code and having everything in one place, readability of your code will be improved.

1. Allows the possibility of tuning preprocessing choices for better test predictions.

Do you know what the effect is of the different preprocessing choices you've made on your model predictions? Testing the effect of preprocessing is often done manually, by making a change and rerunning the notebook. This is slow and inconvenient to do, prone to errors, and it's very difficult to perform a proper workflow evaluation like this. Pipelines can automate the tuning and evaluation of your preprocessing workflow and help to answer questions like:

* 1. Do particular preprocessing steps actually benefit the model?
  2. For categorical encoding (e.g. OneHotEncoder), does grouping smaller infrequent classes help?
  3. What parameters are best for numerical transforms (e.g. power transforms)?
  4. Does dropping certain columns help predictions?

unfold\_lessHide code

In [3]:

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

sample = pd.read\_csv('/kaggle/input/titanic/gender\_submission.csv')

train = pd.read\_csv('/kaggle/input/titanic/train.csv', index\_col = 'PassengerId')

trainX = train.drop(['Survived'], axis=1)

trainy = train['Survived']

test = pd.read\_csv('/kaggle/input/titanic/test.csv', index\_col = 'PassengerId')

1.3 Example data - survial on the Titanic

To demonstrate the use of pipelines to perform data preprocessing, we'll use the titanic dataset. Here we need to predict whether a passenger survived based on the information provided (Age, Sex, Name, etc.).

unfold\_lessHide code

In [4]:

Image("../input/pipeline-diagrams/titanic.png",width=800,height=500)

Out[4]:



unfold\_moreShow hidden code

Out[5]:

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

linkcode

The variables 'Sex', 'Cabin' and 'Embarked' are all string (object) variables, and we'll definitely need to do something with them (i.e. encoding) before a numerical model can be used. There is potentially some information in 'Ticket', 'Name' and 'Cabin' that could be recovered through feature engineering, such as the title ('Mr.', 'Miss.', etc.).

What about missing values?

print('Number of missing values per column')

pd.concat([trainX.isna().sum(0),test.isna().sum(0)], axis=1).rename({0: 'train', 1:'test'}, axis=1)

Number of missing values per column

Out[6]:

|  | train | test |
| --- | --- | --- |
| Pclass | 0 | 0 |
| Name | 0 | 0 |
| Sex | 0 | 0 |
| Age | 177 | 86 |
| SibSp | 0 | 0 |
| Parch | 0 | 0 |
| Ticket | 0 | 0 |
| Fare | 0 | 1 |
| Cabin | 687 | 327 |
| Embarked | 2 | 0 |

linkcode

The analysis shows that 'Age' and 'Cabin' have missing values in both train and test datasets, whereas 'Embarked' and 'Fare' have missing values in the train and test datasets respectively. These features will need to be imputed.

Additionally, we may wish to:

* drop some columns
* create new columns (i.e. feature engineering)
* scale the data (e.g. normalization, standardization or a power transform)

Finally, other data transformations may also be useful (e.g. principal component analysis, or PCA).

With a pipeline we can wrap all of this (and much more!) in the one estimator.

2. Scaling and transforming numerical data

Let's begin with the simplest pipeline: a data processing step and a model (actually, the simplest pipeline we can construct has just one step, but that's not much use). Let's build a model on the titanic dataset to predict 'Surived' based on some of the numerical features: 'Pclass', 'SibSp', 'Parch', 'Fare'. These features have no missing values and require no encoding. Thus, we can apply a power transform directly to these variables directly, and then model the transformed data directly.

In [7]:

plt.figure(figsize=(16,4))

plt.subplots\_adjust(wspace=0.4)

for i,x **in** enumerate(['Pclass','SibSp', 'Parch', 'Fare']):

plt.subplot(1,4,i+1)

sns.histplot(train[x])

group\_mean = train.groupby(train[x])['Survived'].mean()

ax = plt.gca()

ax2=ax.twinx()

sns.regplot(x=group\_mean.index, y=list(group\_mean),lowess=True,scatter=False, color='red')

if i == 0:

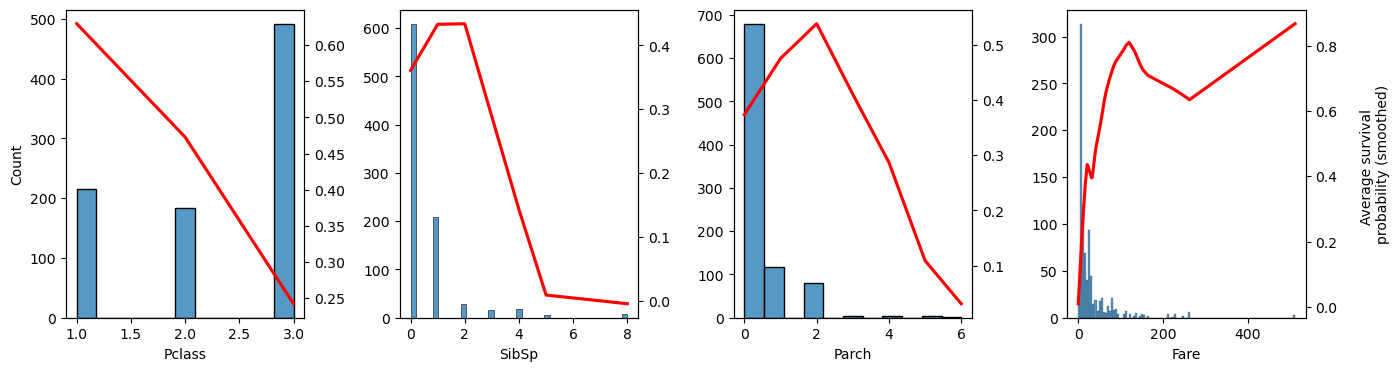
ax.set\_ylabel('Count')

else:

ax.set\_ylabel('')

if i == 3:

ax2.set\_ylabel('**\n**Average survival **\n**probability (smoothed)')



It is evident from histograms of the features we selected that these four variables are skewed. Normalization of these variables could help improve predictions. Sklearn has a transformer (PowerTransformer) that can perform this for us. After transforming 'Fare', the distribution is more even, and it looks like the error in the logistic regression (just for this variable) is reduced.

In [8]:

from sklearn.preprocessing import PowerTransformer

f, axes = plt.subplots(2, 2, gridspec\_kw={'height\_ratios': [3,1]},

figsize=(12,6))

\_=sns.regplot(x=train['Fare'], y=trainy, logistic=True, line\_kws={'color': 'red'},

ax=axes[0,0])

\_=sns.regplot(x=PowerTransformer().fit\_transform(train[['Fare']]), y=trainy, logistic=True,

line\_kws={'color': 'red'},

ax=axes[0,1])

\_=plt.gca().set\_xlabel('Transformed Fare')

\_=sns.histplot(x=train['Fare'],

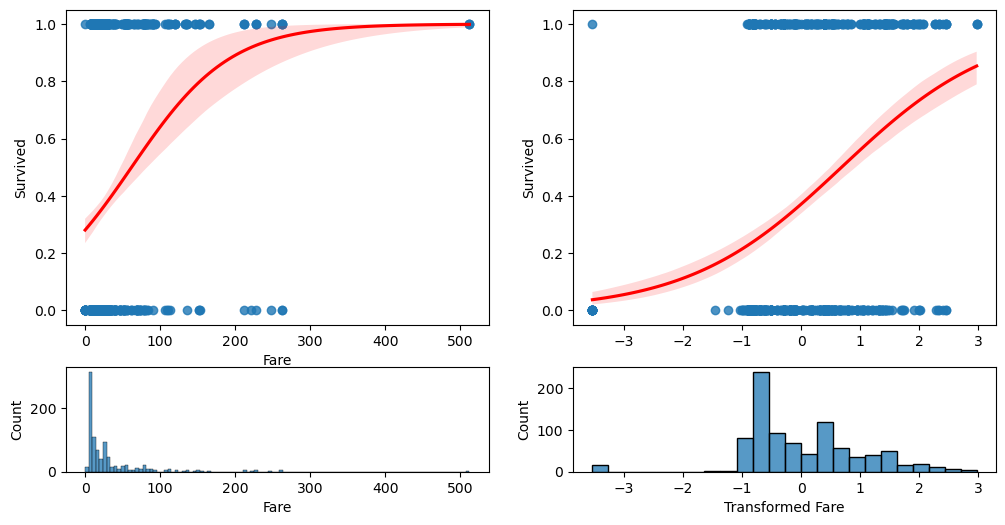
ax = axes[1,0])

\_=plt.gca().set\_xlabel('Fare')

\_=sns.histplot(x=PowerTransformer().fit\_transform(train[['Fare']]).ravel(),

ax = axes[1,1])

\_=plt.gca().set\_xlabel('Transformed Fare')



Now, we can test the effect of applying the power transformation on the four variables before fitting a logistic regression by first fitting a LogisticRegression on the raw data. Call cross\_validate to get an estimate of the cross-validated error from using logistic regression. We use estimator=LogisticRegression():

In [9]:

from sklearn.linear\_model import LogisticRegression

from sklearn.model\_selection import cross\_validate, StratifiedKFold, KFold

cv\_mean\_acc = cross\_validate(estimator=LogisticRegression(),

scoring = 'accuracy',

cv = StratifiedKFold(n\_splits=5, shuffle=True, random\_state=123),

X = train[['Pclass','SibSp', 'Parch', 'Fare']], y=trainy)['test\_score'].mean()

print(f"Average cross-validated accuracy from logistic regression on raw data: **{**cv\_mean\_acc**:**.3f**}**")

Average cross-validated accuracy from logistic regression on raw data: 0.686

To define a pipeline, we just put the steps into a list and call the Pipeline constructor:

In [10]:

from sklearn.pipeline import Pipeline

simplest\_pipeline = Pipeline(steps = [('normalize', PowerTransformer()),

('lr\_model', LogisticRegression())])

Each step has a name (here 'normalize' and 'lr\_model') and the associated transformer/estimator. The names can be anything, but we'll see later that we use these names to set and retrieve parameters for each step of the pipeline using the set\_params() and get\_params() methods, and also through cross-validation schemes like GridSearchCV.

To fit the logistic regression on the transformed data, we pass the pipeline as the estimator in cross\_validate along with the raw data, rather than the model instance (i.e. LogisticRegression) and the transformed data:

In [11]:

skf = StratifiedKFold(n\_splits=3, shuffle=True, random\_state=124)

cv\_mean\_acc\_pipeline = cross\_validate(estimator=simplest\_pipeline,

scoring = 'accuracy',

cv = skf,

X = train[['Pclass','SibSp', 'Parch', 'Fare']], y=trainy)['test\_score'].mean()

print(f"Average cross-validated accuracy from logistic regression with transformed data: **{**cv\_mean\_acc\_pipeline**:**.3f**}**")

Average cross-validated accuracy from logistic regression with transformed data: 0.708

Success! The cross-validated accuracy is improved when we include a power transformation before the logistic regression.

We could apply other numerical transformations (such as principal component analysis: sklearn.decomposition.PCA, and scaling, e.g. sklearn.preprocessing.StandardScaler) in a similar way.

**NB:** The pipeline workflow allows us to fit and transform the training and test data *within* the cross-validation scheme used by cross\_validate, whereas cross validating the logistic regression model directly and passing the pre-transformed data as parameters 'X' and 'y' in the call to cross\_validate can result in data leakage.

In [12]:

from sklearn.impute import SimpleImputer

from sklearn.preprocessing import OrdinalEncoder, OneHotEncoder, StandardScaler, FunctionTransformer

from sklearn.compose import ColumnTransformer

from sklearn.base import BaseEstimator, TransformerMixin

from sklearn.pipeline import FeatureUnion

unfold\_moreShow hidden code

3. Imputation

In a general sense, imputation can be thought of as the "filling in" of missing values by some scheme.

Imputation strategies can range from simple to very complex. For this example, we will impute in a simple way by filling in with the mean value of non-missing data.

Take the 'Age' column as an example. There are 177 and 86 missing rows in the training and test datasets, and the mean from the non-missing training data is 29.7 years:

In [14]:

print(f'Mean of non-missing Age column in training dataset: **{**train["Age"].mean()**:**.1f**}**')

print(f'Mean of non-missing Age column in test dataset: **{**test["Age"].mean()**:**.1f**}**')

Mean of non-missing Age column in training dataset: 29.7

Mean of non-missing Age column in test dataset: 30.3

3.1 Direct imputation

A direct (naive) way to do this is to fill in the values using the index values from a call to isna():

train\_copy = train.copy()

train\_copy.loc[train\_copy['Age'].isna(), 'Age'] = train\_copy['Age'].mean()

To apply this to the test dataset, we would need to write something like:

test[test['Age'].isna(), 'Age'] = train['Age'].mean()`

However, this violates the DRY coding principle in that we are repeating the same statement but with different datasets. Errors can quickly creep in when with this, either when the original code is changed (e.g. we decide to infill with a median), or we forget to change train to test when modifying the copied code. We can do better than this!

3.2 Transformer-based method of imputation

The direct (procedural) way of imputing as described above can't really be implemented in a pipeline. Since pipelines are built from scikit-learn transformers, we need a transformer-based way of filling in the missing values. Luckily, scikit-learn has a transformer that does exactly this, namely SimpleImputer. We use this by calling fit on the training data, then transform on the training data and then transform on the test data. Alternatively, the first two steps can be combined with a call to fit\_transform:

In [15]:

from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='mean')

imputed\_train\_Age = imputer.fit\_transform(train[['Age']])

imputed\_test\_Age = imputer.transform(test[['Age']])

The output imputed\_train\_Age and imputed\_test\_Age are exactly what we want, i.e. arrays (or columns) of 'Age' with missing values filled in. Notice that we use the same object (imputer) to apply the imputation to both training and test datasets, so we are not violating the DRY principle.

In [16]:

print(f'Mean of imputed\_train\_Age: **{**imputed\_train\_Age.mean()**:**.1f**}**')

print(f'Mean of imputed\_test\_Age: **{**imputed\_test\_Age.mean()**:**.1f**}**')

Mean of imputed\_train\_Age: 29.7

Mean of imputed\_test\_Age: 30.2

3.3 Imputation pipeline

Now that imputation is set up with a transformer, we can include this (either imputer or SimpleImputer(strategy='mean') directly) as a pipeline step, e.g.

imputation\_pipeline = Pipeline(steps=[('impute', SimpleImputer(strategy='mean')),

('model', LogisticRegression())])

imputation\_pipeline.fit(train[['Age']], trainy)

Notice that we only pass through train[['Age']] at this point as each step of the pipeline is applied to the entire data set, unless we use a ColumnTransformer (see below).

4. Encoding categorical data

The two categorical columns 'Sex' and 'Embarked' will need to be encoded for use in most models

One way of doing this is with pandas [get\_dummies()](https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html) function (in theory we could do it manually as well...), but since we are trying to set up a pipeline it is best to consider a scikit-learn transformer approach to doing this.

4.1 Encoding with the OneHotEncoder transform

Applying the OneHotEncoder transform directly is fairly straight-forward. I usually setting the parameter handle\_unknown to 'ignore', because it there's a category in the test data that isn't in the training data, the pipeline won't throw an error.

In [17]:

from sklearn.preprocessing import OneHotEncoder

one = OneHotEncoder(sparse\_output=False,

handle\_unknown='ignore')

pd.concat([train['Sex'], pd.DataFrame(one.fit\_transform(train[['Sex']]),

index=train.index)], axis=1)

Out[17]:

|  | Sex | 0 | 1 |
| --- | --- | --- | --- |
| PassengerId |  |  |  |
| 1 | male | 0.0 | 1.0 |
| 2 | female | 1.0 | 0.0 |
| 3 | female | 1.0 | 0.0 |
| 4 | female | 1.0 | 0.0 |
| 5 | male | 0.0 | 1.0 |
| ... | ... | ... | ... |
| 887 | male | 0.0 | 1.0 |
| 888 | female | 1.0 | 0.0 |
| 889 | female | 1.0 | 0.0 |
| 890 | male | 0.0 | 1.0 |
| 891 | male | 0.0 | 1.0 |

891 rows × 3 columns

Here, the 'Sex' column has been split into two indicator variables, one each for male and female.

Similarly to the imputation section above, we could set up a pipeline that operates on just the categorical columns:

encoding\_pipeline = Pipeline(steps=[('encode', OneHotEncoder(handle\_unknown='ignore')),

('model', LogisticRegression())])

encoding\_pipeline.fit(train[['Sex', 'Embarked']], trainy)

In reality, though, we probably want to impute some columns, encode other columns, and maybe do nothing to other columns. The example pipelines we have set up so far only work on specific columns. In order to selectively apply transformers to different columns we need ColumnTransformer:

5. ColumnTransformer - applying different transformations to different columns

In any practical application, there are going to be different types of columns, which need different treatments. ColumnTransformer selectively applies transformers to different columns. We saw that the basic syntax of pipelines in scikit-learn was a list of steps. For each step we specify its name (which can be anything we like) and the transformer itself in a tuple. [The syntax](https://scikit-learn.org/stable/modules/generated/sklearn.compose.ColumnTransformer.html) for ColumnTransformer is similar but we also need to specify the relevant columns for each step as well in the third position of the tuple.

In [18]:

from sklearn.compose import ColumnTransformer

multicolumn\_prep = ColumnTransformer([('impute',

SimpleImputer(strategy='mean'),

['Age', 'Fare']),

('encode',

OneHotEncoder(handle\_unknown='ignore'),

['Sex', 'Embarked']),

],

remainder='passthrough')

multicolumn\_prep

Out[18]:

ColumnTransformer

impute

SimpleImputer

encode

OneHotEncoder

remainder

passthrough

Two things to note about the ColumnTransformer defined here are that: 1) we don't need to impute 'Embarked' as the encoding will encode missing values ('nan') as a separate category, and 2) any column not specified in the transformer list (such as 'Pclass') will 'passthrough' the ColumnTransformer as specified by the remainder argument, i.e. these variables will stay untransformed.

Now, we can include this as the preprocessing step in a pipeline.

In [19]:

ct\_pipeline = Pipeline([('preprocessing', multicolumn\_prep),

('lr\_model', LogisticRegression(max\_iter = 2000))])

ct\_pipeline

Out[19]:

Pipeline

preprocessing: ColumnTransformer

impute

SimpleImputer

encode

OneHotEncoder

remainder

passthrough

LogisticRegression

As our pipeline gets more and more complicated, it is worth pointing out at this point that we can get an interactive flowchart representation of the pipeline through the set\_config function:

In [20]:

from sklearn import set\_config

set\_config(display="diagram")

ct\_pipeline

Out[20]:

Pipeline

preprocessing: ColumnTransformer

impute

SimpleImputer

encode

OneHotEncoder

remainder

passthrough

LogisticRegression

Now, let's get an estimate of the accuracy from this preprocessing step combined with the logistic regression:

In [21]:

ct\_cv\_res = cross\_validate(estimator = ct\_pipeline,

X = trainX.drop(['Name', 'Ticket', 'Cabin'],

axis=1),

y = trainy,

cv = skf,

scoring = 'accuracy')['test\_score'].mean()

print(f"Average cross-validated accuracy from**\n**column transformer pipeline: **{**ct\_cv\_res**:**.3f**}**")

Average cross-validated accuracy from

column transformer pipeline: 0.799

Pretty good! These four variables encoded properly provide a pretty good model, better than the transformed numerical model in section 2.

6. Replacing a column with a custom function

The preprocessing transformers available with sklearn.preprocessing can do lots of things, but not everything. Processing of string (object) feature will in general require a custom function to extract and transform the relevant information. As an example, let's process the 'Cabin' feature.

In [22]:

train['Cabin']

Out[22]:

PassengerId

1 NaN

2 C85

3 NaN

4 C123

5 NaN

...

887 NaN

888 B42

889 NaN

890 C148

891 NaN

Name: Cabin, Length: 891, dtype: object

This column contains either the cabin number, e.g. 'C85', or NaN if the passenger wasn't in a cabin. The cabin number looks to consist of a deck 'A', 'B', 'C' and so on, and a number. Under the assumption that the deck could contain predictive information but the number on the deck won't, let's consider extracting the initial letter from this feature.

6.1 List-comprehension approach

Extracting the deck letter is slightly hampered by the fact that NaN is considered a number (i.e. a float) but the cabin ID 'C85' is a string. We can use a conditional list comprehension to get what we want:

In [23]:

[x[0] if type(x) == str else 'None' for x **in** train['Cabin']][:15]

Out[23]:

['None',

'C',

'None',

'C',

'None',

'None',

'E',

'None',

'None',

'None',

'G',

'C',

'None',

'None',

'None']

We can process this into a kind of ordinal encoding, so that it can be used directly, as follows:

In [24]:

[ord(x[0]) - ord('A') + 1 if type(x) == str else 0 for x **in** train['Cabin']][:15]

Out[24]:

[0, 3, 0, 3, 0, 0, 5, 0, 0, 0, 7, 3, 0, 0, 0]

Here, 0 stands for no cabin, 1 for 'A', etc.

To incorporate this text extraction step into a pipeline we need to use a transformer rather than a function (or list comprehension). Without a pipeline we could use something like:

train['Cabin'] = [x[0] if type(x) == str else 'None' for x in train['Cabin']]

but this approach is discouraged as discussed in the introduction section.

6.2 FunctionTransformer approach

In the same way that we replaced a direct approach to imputation with a scikit-learn transformation approach in section 3.2, we replace the list-comprehension method of extracting the deck letter from 'Cabin' with a FunctionTransformer, which allows us to define a transformer based on an arbitrary function.

In [25]:

extract\_cabin = FunctionTransformer(func = lambda col: np.array([ord(x[0]) - ord('A') + 1 \

if type(x) == str else \

0 for x **in** col])[:,np.newaxis])

extract\_cabin.fit\_transform(train['Cabin'])[:15]

Out[25]:

array([[0],

[3],

[0],

[3],

[0],

[0],

[5],

[0],

[0],

[0],

[7],

[3],

[0],

[0],

[0]])

You may need to playing around a bit with the functional form of the transformation so that it outputs an array (column) rather than a 1-D list. To use the extract\_cabin transformer in a ColumnTransformer we need an array so that it can be column-bound with the rest of the output from the transformers. The list comprehension that we developed operates on a single column (col), and returns a length-n list.

In the FunctionTransformer, we cast the list as an array and add an extra dimension (using np.newaxis) to get an (n, 1) shaped array, which will work with the rest of the steps from ColumnTransformer.

Now, we can combine this with the imputation and encoding in ColumnTransformer:

In [26]:

multicolumn\_prep\_with\_cabin = ColumnTransformer([('impute',

SimpleImputer(strategy='mean'),

['Age', 'Fare']),

('cabin\_extract',

extract\_cabin,

'Cabin'),

('encode',

OneHotEncoder(handle\_unknown='ignore',

sparse\_output=False),

['Sex', 'Embarked']),

],

remainder='passthrough')

multicolumn\_prep\_with\_cabin

Out[26]:

ColumnTransformer

impute

SimpleImputer

cabin\_extract

FunctionTransformer

encode

OneHotEncoder

remainder

passthrough

Notice that the specified column for the extract\_cabin transformer is 'Cabin' rather than ['Cabin'], and this is due to the way the FunctionTransformer is set up, if we had a true array function (like the [example in the sklearn documentation](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.FunctionTransformer.html)), we could have the columns to be transformed in a list as with the other steps in the ColumnTransformer. See also [this discussion on stack overflow](https://stackoverflow.com/questions/56298242/valueerror-input-array-dimensions-not-right-for-countvectorizer/56299794#56299794) about this problem.

**Tip:** it is best to use an array function in FunctionTransformer

Now, let's have a look to see if this improves our modelling:

In [27]:

cv\_with\_cabin = cross\_validate(estimator = Pipeline([('preprocessing', multicolumn\_prep\_with\_cabin),

('lr', LogisticRegression(max\_iter=2000))]),

X = trainX.drop(['Name', 'Ticket'],

axis=1),

y = trainy,

cv = skf,

scoring = 'accuracy')['test\_score'].mean()

print(f"Average cross-validated accuracy including 'Cabin' feature engineering: **{**cv\_with\_cabin**:**.3f**}**")

Average cross-validated accuracy including 'Cabin' feature engineering: 0.810

6.3 Custom transformer approach

More flexibility can be obtained by using custom transformers, although it is often a little bit more work to set up the class. Most of the work we need to do is in writing the fit() and transform() methods, whch is where most of the action happens.

We can get most of what we need to make a pipeline-ready transformer by inheriting from TransformerMixin and BaseEstimator. This includes \_\_init\_\_(), fit\_transform() (which just runs fit() and then transform()), and get\_params() and set\_params() (which we need if we're using the pipeline with GridSearchCV, etc).

If we'd like to specify a parameter for the transformation, we need to override the default \_\_init\_\_() method. [Setting parameters](https://scikit-learn.org/stable/developers/develop.html#parameters-and-init) in the \_\_init\_\_() method means we can set these parameters in the call to the class (like the way we specified mean imputation using SimpleImputer(strategy='mean') above).

In a general sense, the fit() method looks at the input ('X') and fits parameters, such as minimum and maximum for a scaling transformer, or categories for an encoding transformer. Note that fit() methods of scikit-learn transformers need to end with return self to work well with the rest of the scikit-learn architecture.

The transform() method is where we actually apply the transformation to the input data (both train and test).

6.3.1 Extract title from 'Name'

Although the names of passengers might not be directly useful, one idea is to extract the title (Mr, Miss, etc.) from the name, which we can encode, and perhaps this may be useful to a model.

unfold\_moreShow hidden code

Out[28]:



*Image courtesy of DALL-E*

We pass in a 'min\_relative\_frequency' parameter into the call to the class (through \_\_init\_\_()), which thresholds the titles, replacing uncommon titles (such as 'Jonkheer') with 'Rare/Unknown'

In [29]:

from sklearn.base import BaseEstimator, TransformerMixin

class ExtractTitle(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, min\_relative\_freq):

self.min\_relative\_freq = min\_relative\_freq

def fit(self, X, y=None):

from collections import Counter

title\_freq = Counter()

titles = [x.split(',')[1].split('.')[0].strip() for x **in** X['Name']]

title\_freq.update(titles)

self.common\_titles = [x[0] for x **in** title\_freq.items() if x[1] > X.shape[0]\*self.min\_relative\_freq]

return self

def transform(self, X, y=None):

title = [x.split(',')[1].split('.')[0].strip() for x **in** X['Name']]

X\_copy = X.copy()

X\_copy['Title'] = [x if x **in** self.common\_titles else 'Rare/Unknown' for x **in** title]

return X\_copy.drop(['Name'], axis=1)

Rather than combining our transformer with ColumnTransformer as we did in section 6.2, to replace the 'Name' column with 'Title', we add the processed column onto the dataframe and then drop the original column directly in the transform() method. The custom transformer approach is more flexible then the FunctionTransformer/ColumnTransformer combination, but the transformers usually end up being more specific to the task at hand, and hence less reusable.

In [30]:

xtract\_title = ExtractTitle(min\_relative\_freq = 0.2)

xtract\_title.fit\_transform(train)

Out[30]:

|  | Survived | Pclass | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked | Title |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerId |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 3 | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S | Mr |
| 2 | 1 | 1 | female | 38.0 | 1 | 0 | PC 17599 | 71.2833 | C85 | C | Rare/Unknown |
| 3 | 1 | 3 | female | 26.0 | 0 | 0 | STON/O2. 3101282 | 7.9250 | NaN | S | Miss |
| 4 | 1 | 1 | female | 35.0 | 1 | 0 | 113803 | 53.1000 | C123 | S | Rare/Unknown |
| 5 | 0 | 3 | male | 35.0 | 0 | 0 | 373450 | 8.0500 | NaN | S | Mr |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 887 | 0 | 2 | male | 27.0 | 0 | 0 | 211536 | 13.0000 | NaN | S | Rare/Unknown |
| 888 | 1 | 1 | female | 19.0 | 0 | 0 | 112053 | 30.0000 | B42 | S | Miss |
| 889 | 0 | 3 | female | NaN | 1 | 2 | W./C. 6607 | 23.4500 | NaN | S | Miss |
| 890 | 1 | 1 | male | 26.0 | 0 | 0 | 111369 | 30.0000 | C148 | C | Mr |
| 891 | 0 | 3 | male | 32.0 | 0 | 0 | 370376 | 7.7500 | NaN | Q | Mr |

891 rows × 11 columns

Here the 'Name' column has been replaced with the 'Title' column.

6.3.2 Discretizing the 'Fare' numerical column

As another example of using a custom transformer, we can bin numerical columns. Rather than having 'Fare' as a numerical (continuous) variable, we can discretize (bin) it, to convert it into an ordinal column. Since 'Fare' is highly skewed, we bin this in quantile-space, using the pd.qcut function

In [31]:

class QCutFare(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, n\_bins):

self.n\_bins = n\_bins

def fit(self, X, y=None):

return self

def transform(self, X, y=None):

transformed = pd.qcut(X['Fare'], self.n\_bins, labels=False)

transformed[transformed.isna()] = 0

transformed\_series = pd.Series(transformed,

name=f'QCut**{**self.n\_bins**}**\_Fare',

index=X.index)

X\_copy = X.copy()

return pd.concat([X\_copy, transformed\_series], axis=1).drop(['Fare'], axis=1)

QCutFare(13).fit\_transform(train)

Out[31]:

|  | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Cabin | Embarked | QCut13\_Fare |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerId |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | NaN | S | 1 |
| 2 | 1 | 1 | Cumings, Mrs. John Bradley (Florence Briggs Th... | female | 38.0 | 1 | 0 | PC 17599 | C85 | C | 11 |
| 3 | 1 | 3 | Heikkinen, Miss. Laina | female | 26.0 | 0 | 0 | STON/O2. 3101282 | NaN | S | 3 |
| 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35.0 | 1 | 0 | 113803 | C123 | S | 10 |
| 5 | 0 | 3 | Allen, Mr. William Henry | male | 35.0 | 0 | 0 | 373450 | NaN | S | 3 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 887 | 0 | 2 | Montvila, Rev. Juozas | male | 27.0 | 0 | 0 | 211536 | NaN | S | 5 |
| 888 | 1 | 1 | Graham, Miss. Margaret Edith | female | 19.0 | 0 | 0 | 112053 | B42 | S | 9 |
| 889 | 0 | 3 | Johnston, Miss. Catherine Helen "Carrie" | female | NaN | 1 | 2 | W./C. 6607 | NaN | S | 7 |
| 890 | 1 | 1 | Behr, Mr. Karl Howell | male | 26.0 | 0 | 0 | 111369 | C148 | C | 9 |
| 891 | 0 | 3 | Dooley, Mr. Patrick | male | 32.0 | 0 | 0 | 370376 | NaN | Q | 1 |

891 rows × 11 columns

The 'Fare' column is replaced with the 'QCut13\_Fare' column, which specifies which quantile-bin the fare belongs to.

7. Feature engineering - adding a column

Adding columns can be a useful way of extracting extra information from a dataset. Creating new columns containing summaries (mean, variance, etc.) of multiple columns can provide a lot of predictive information. The best way to do this might be to calculate the statistic in a new column, but to also retain the columns (ColumnTransformer-based approaches that we have used so far will replace columns with the transformation).

Custom transformers can achieve this by concatenating columns onto the 'X' matrix in the transform step. Let's take a closer look at the 'Age' column:

In [32]:

np.unique(train['Age'])

Out[32]:

array([ 0.42, 0.67, 0.75, 0.83, 0.92, 1. , 2. , 3. , 4. ,

5. , 6. , 7. , 8. , 9. , 10. , 11. , 12. , 13. ,

14. , 14.5 , 15. , 16. , 17. , 18. , 19. , 20. , 20.5 ,

21. , 22. , 23. , 23.5 , 24. , 24.5 , 25. , 26. , 27. ,

28. , 28.5 , 29. , 30. , 30.5 , 31. , 32. , 32.5 , 33. ,

34. , 34.5 , 35. , 36. , 36.5 , 37. , 38. , 39. , 40. ,

40.5 , 41. , 42. , 43. , 44. , 45. , 45.5 , 46. , 47. ,

48. , 49. , 50. , 51. , 52. , 53. , 54. , 55. , 55.5 ,

56. , 57. , 58. , 59. , 60. , 61. , 62. , 63. , 64. ,

65. , 66. , 70. , 70.5 , 71. , 74. , 80. , nan])

We see that there are a number of fractional ages for babies below 12 months old, and, for some reason, there are a number of half ages (e.g. 20.5). It may be that people with fractional ages had more or less chance of survival, so let's engineer an indicator variable for this feature. We keep the 'Age' column itself as it is one of the more useful features.

7.1 Add a fractional-age indicator column

7.1.1 List-comprehension approach

Essentially, we can create a fractional-age indicator with the list comprehension [int(x)!=x for x in train['Age']], except for the fact we have missing values. Count missing ages ('nan') as non-fractional ages:

In [33]:

fractional\_age = [0 if np.isnan(x) else 1\*(int(x)!=x) for x **in** train['Age']]

ismissing\_age = [1 if np.isnan(x) else 0 for x **in** train['Age']]

pd.DataFrame({'Age':train['Age'], 'FracAgeInd': fractional\_age, 'AgeMissInd':ismissing\_age}).iloc[105:115,:]

Out[33]:

|  | Age | FracAgeInd | AgeMissInd |
| --- | --- | --- | --- |
| PassengerId |  |  |  |
| 106 | 28.0 | 0 | 0 |
| 107 | 21.0 | 0 | 0 |
| 108 | NaN | 0 | 1 |
| 109 | 38.0 | 0 | 0 |
| 110 | NaN | 0 | 1 |
| 111 | 47.0 | 0 | 0 |
| 112 | 14.5 | 1 | 0 |
| 113 | 22.0 | 0 | 0 |
| 114 | 20.0 | 0 | 0 |
| 115 | 17.0 | 0 | 0 |

And we can combine these as

In [34]:

pd.DataFrame({'Age':train['Age'],

'FractionalAge':[0 if np.isnan(x) else 1\*(int(x)!=x) for x **in** train['Age']]}).loc[105:115,:]

Out[34]:

|  | Age | FractionalAge |
| --- | --- | --- |
| PassengerId |  |  |
| 105 | 37.0 | 0 |
| 106 | 28.0 | 0 |
| 107 | 21.0 | 0 |
| 108 | NaN | 0 |
| 109 | 38.0 | 0 |
| 110 | NaN | 0 |
| 111 | 47.0 | 0 |
| 112 | 14.5 | 1 |
| 113 | 22.0 | 0 |
| 114 | 20.0 | 0 |
| 115 | 17.0 | 0 |

7.1.2 Transformer approach

Using FunctionTransformer and ColumnTransformer replaces columns with their transformations, so these are not useful for this.

As far as I'm aware, we need to create a custom transformer to add columns to a dataset in a pipeline. A  does something similar in section 3 of that article. This is similar to what we did in section 6.3, except here we don't want to drop the orignal column.

Here's a translation of the list-comprehension approach into a scikit-learn transformer class. In a similar way to the way we added 'Title' onto the dataframe in the transform() method of ExtractTitle, we append 'FractionalAge' in transform():

In [35]:

from sklearn.base import TransformerMixin, BaseEstimator

class AddFractionalAgeColumn(TransformerMixin, BaseEstimator):

def fit(self, X, y=None):

return self

def transform(self, X, y=None):

return pd.concat([X, pd.Series([0 if np.isnan(x) else 1\*(int(x)!=x) for x **in** X['Age']],

index=X.index,

name = 'FractionalAge')], axis=1)

Let's see the transformer in action:

In [36]:

AddFractionalAgeColumn().fit\_transform(train).iloc[105:115,:]

Out[36]:

|  | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked | FractionalAge |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerId |  |  |  |  |  |  |  |  |  |  |  |  |
| 106 | 0 | 3 | Mionoff, Mr. Stoytcho | male | 28.0 | 0 | 0 | 349207 | 7.8958 | NaN | S | 0 |
| 107 | 1 | 3 | Salkjelsvik, Miss. Anna Kristine | female | 21.0 | 0 | 0 | 343120 | 7.6500 | NaN | S | 0 |
| 108 | 1 | 3 | Moss, Mr. Albert Johan | male | NaN | 0 | 0 | 312991 | 7.7750 | NaN | S | 0 |
| 109 | 0 | 3 | Rekic, Mr. Tido | male | 38.0 | 0 | 0 | 349249 | 7.8958 | NaN | S | 0 |
| 110 | 1 | 3 | Moran, Miss. Bertha | female | NaN | 1 | 0 | 371110 | 24.1500 | NaN | Q | 0 |
| 111 | 0 | 1 | Porter, Mr. Walter Chamberlain | male | 47.0 | 0 | 0 | 110465 | 52.0000 | C110 | S | 0 |
| 112 | 0 | 3 | Zabour, Miss. Hileni | female | 14.5 | 1 | 0 | 2665 | 14.4542 | NaN | C | 1 |
| 113 | 0 | 3 | Barton, Mr. David John | male | 22.0 | 0 | 0 | 324669 | 8.0500 | NaN | S | 0 |
| 114 | 0 | 3 | Jussila, Miss. Katriina | female | 20.0 | 1 | 0 | 4136 | 9.8250 | NaN | S | 0 |
| 115 | 0 | 3 | Attalah, Miss. Malake | female | 17.0 | 0 | 0 | 2627 | 14.4583 | NaN | C | 0 |

The output in the 'FractionalAge' column is identical to the output obtained from the list-comprehension approach in section 7.1.1.

7.2 Column aggregation - Family size

Calculating summaries (mean, variance, count, etc.) of multiple columns can be a powerful way to engineer new features. We can calculate the size of the family on board for each passenger by adding 'SibSp' and 'Parch' (plus one for the passenger itself). A transformer adding columns specified in the 'columns' parameter follows:

In [37]:

class AddSumOfColumnsColumn(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, columns, constant=0, name = 'AddedColumns'):

self.columns = columns

self.constant = constant

self.name = name

def fit(self, X, y=None):

return self

def transform(self, X, y=None):

X\_copy = X.copy()

added\_series = X[self.columns].sum(1) + self.constant

added\_series.name = self.name

return pd.concat([X, added\_series], axis=1)

In [38]:

AddSumOfColumnsColumn(columns=['SibSp','Parch'],

constant=1,

name='FamilySize').fit\_transform(train)

Out[38]:

|  | Survived | Pclass | Name | Sex | Age | SibSp | Parch | Ticket | Fare | Cabin | Embarked | FamilySize |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerId |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 0 | 3 | Braund, Mr. Owen Harris | male | 22.0 | 1 | 0 | A/5 21171 | 7.2500 | NaN | S | 2 |
| 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 | 1 |
| 4 | 1 | 1 | Futrelle, Mrs. Jacques Heath (Lily May Peel) | female | 35.0 | 1 | 0 | 113803 | 53.1000 | C123 | S | 2 |
| 5 | 0 | 3 | Allen, Mr. William Henry | male | 35.0 | 0 | 0 | 373450 | 8.0500 | NaN | S | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 887 | 0 | 2 | Montvila, Rev. Juozas | male | 27.0 | 0 | 0 | 211536 | 13.0000 | NaN | S | 1 |
| 888 | 1 | 1 | Graham, Miss. Margaret Edith | female | 19.0 | 0 | 0 | 112053 | 30.0000 | B42 | S | 1 |
| 889 | 0 | 3 | Johnston, Miss. Catherine Helen "Carrie" | female | NaN | 1 | 2 | W./C. 6607 | 23.4500 | NaN | S | 4 |
| 890 | 1 | 1 | Behr, Mr. Karl Howell | male | 26.0 | 0 | 0 | 111369 | 30.0000 | C148 | C | 1 |
| 891 | 0 | 3 | Dooley, Mr. Patrick | male | 32.0 | 0 | 0 | 370376 | 7.7500 | NaN | Q | 1 |

891 rows × 12 columns

**Tip:** it is easiest to add columns in a pipeline using a custom transformer.

7.3 Alternative method: FeatureUnion

Obviously these classes are very specific to the task at hand. Extending this approach, we might like to create more general transformer classes that add columns according to different transformations and working on different (or variable) columns. A more general way of doing this could be to use FeatureUnion, which creates several features from consecutive passes through a data frame.

[As described on stack overflow](https://stackoverflow.com/q/55604249), FeatureUnion differs from ColumnTransformer in that each step of FeatureUnion processes the entire dataframe, whereas each step of ColumnTransformer processes a disjoint subset of columns.

8. Drop columns

Notice that in the pipelines we have constructed so far, we have dropped the columns we are not interested in (like 'Name' and 'Ticket' in section 6.2) before passing the data through into the pipeline. Some custom transformers drop a column in the transform method

We could alternatively incorporate column dropping into the pipeline itself.

Why would we want to do this? Well, we may want to test the effect of dropping columns on our final model using GridSearchCV. Most machine-learning models these days are pretty robust to the presence of uninformative columns, but predictions *can* be improved by pruning off useless columns.

The following transformer does exactly this, and we can use it as a step in the pipeline. We put the column names to drop in the constructor.

In [39]:

class DropColumn(BaseEstimator, TransformerMixin):

def \_\_init\_\_(self, cols=[]):

self.cols = cols

def fit(self, X, y=None):

return self

def transform(self, X, y=None):

return X.drop(self.cols, axis=1)

**NB:** Dropping columns that are specified in a down-stream ColumnTransformer will cause problems. If the ColumnTransformer looks for a column that has been dropped, a ValueError will occur.

9. Named transformer output

Let's combine the transformers we've developed so far into a pipeline:

In [40]:

preprocessing\_pipeline = Pipeline([('extract\_title', ExtractTitle(min\_relative\_freq = 0.2)),

('discretize\_Fare', QCutFare(13)),

('fractional\_Age', AddFractionalAgeColumn()),

('family\_Size', AddSumOfColumnsColumn(columns=['SibSp','Parch'],

constant=1,

name='FamilySize')),

('drop', DropColumn(cols=['Ticket'])),

('prep',ColumnTransformer([('impute',

SimpleImputer(strategy='mean'),

['Age']),

('cabin\_extract',

extract\_cabin,

'Cabin'),

('encode',

OneHotEncoder(handle\_unknown='ignore',

sparse\_output=False),

['Sex', 'Embarked', 'Title']),

],

remainder='passthrough')),

])

preprocessing\_pipeline

Out[40]:

Pipeline

ExtractTitle

QCutFare

AddFractionalAgeColumn

AddSumOfColumnsColumn

DropColumn

prep: ColumnTransformer

impute

SimpleImputer

cabin\_extract

FunctionTransformer

encode

OneHotEncoder

remainder

passthrough

In [41]:

pd.DataFrame(preprocessing\_pipeline.fit\_transform(train.drop(['Survived'], axis=1)))

Out[41]:

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 22.000000 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 1.0 | 0.0 | 1.0 | 0.0 | 2.0 |
| 1 | 38.000000 | 3.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 0.0 | 11.0 | 0.0 | 2.0 |
| 2 | 26.000000 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 3.0 | 0.0 | 0.0 | 3.0 | 0.0 | 1.0 |
| 3 | 35.000000 | 3.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 1.0 | 0.0 | 10.0 | 0.0 | 2.0 |
| 4 | 35.000000 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 3.0 | 0.0 | 1.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 886 | 27.000000 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 2.0 | 0.0 | 0.0 | 5.0 | 0.0 | 1.0 |
| 887 | 19.000000 | 2.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 9.0 | 0.0 | 1.0 |
| 888 | 29.699118 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 3.0 | 1.0 | 2.0 | 7.0 | 0.0 | 4.0 |
| 889 | 26.000000 | 3.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 9.0 | 0.0 | 1.0 |
| 890 | 32.000000 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 |

891 rows × 17 columns

In [42]:

pd.DataFrame(preprocessing\_pipeline.transform(test))

Out[42]:

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 34.500000 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 2.0 | 1.0 | 1.0 |
| 1 | 47.000000 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 3.0 | 1.0 | 0.0 | 0.0 | 0.0 | 2.0 |
| 2 | 62.000000 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 2.0 | 0.0 | 0.0 | 4.0 | 0.0 | 1.0 |
| 3 | 27.000000 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 4.0 | 0.0 | 1.0 |
| 4 | 22.000000 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 3.0 | 1.0 | 1.0 | 5.0 | 0.0 | 3.0 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 413 | 29.699118 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 3.0 | 0.0 | 1.0 |
| 414 | 39.000000 | 3.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 12.0 | 0.0 | 1.0 |
| 415 | 38.500000 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1.0 |
| 416 | 29.699118 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3.0 | 0.0 | 0.0 | 3.0 | 0.0 | 1.0 |
| 417 | 29.699118 | 0.0 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 3.0 | 1.0 | 1.0 | 7.0 | 0.0 | 3.0 |

418 rows × 17 columns

Great! the transformers produce a clean and usable data frame applied to both the training and test data. There is one small problem, though, and that is that the column titles have disappeared. Whereas the column titles are retained with the custom transformers that we defined ourselves:

In [43]:

Pipeline(preprocessing\_pipeline.steps[:-1]).fit\_transform(train.drop(['Survived'], axis=1))

Out[43]:

|  | Pclass | Sex | Age | SibSp | Parch | Cabin | Embarked | Title | QCut13\_Fare | FractionalAge | FamilySize |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerId |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 3 | male | 22.0 | 1 | 0 | NaN | S | Mr | 1 | 0 | 2 |
| 2 | 1 | female | 38.0 | 1 | 0 | C85 | C | Rare/Unknown | 11 | 0 | 2 |
| 3 | 3 | female | 26.0 | 0 | 0 | NaN | S | Miss | 3 | 0 | 1 |
| 4 | 1 | female | 35.0 | 1 | 0 | C123 | S | Rare/Unknown | 10 | 0 | 2 |
| 5 | 3 | male | 35.0 | 0 | 0 | NaN | S | Mr | 3 | 0 | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 887 | 2 | male | 27.0 | 0 | 0 | NaN | S | Rare/Unknown | 5 | 0 | 1 |
| 888 | 1 | female | 19.0 | 0 | 0 | B42 | S | Miss | 9 | 0 | 1 |
| 889 | 3 | female | NaN | 1 | 2 | NaN | S | Miss | 7 | 0 | 4 |
| 890 | 1 | male | 26.0 | 0 | 0 | C148 | C | Mr | 9 | 0 | 1 |
| 891 | 3 | male | 32.0 | 0 | 0 | NaN | Q | Mr | 1 | 0 | 1 |

891 rows × 11 columns

the ColumnTransformer and attached transformers (SimpleImputer, etc.) throw away the column names and return numpy arrays rather than pandas dataframes. This is OK when we pass the array through to a model, but is less than ideal for further data exploration, or indeed interpretation of the model outputs (such as feature importances, etc.). Furthermore, ColumnTransformer reorders the output so that we can't even easily match up the columns in the output data with the column names in the input data.

.

We'll need to rewrite the extract\_cabin transformer from section 6.2 using FunctionTransformer to return a data frame (ignore the warning):

In [44]:

extract\_cabin\_named = FunctionTransformer(func = lambda df: df.apply(lambda col: [ord(x[0]) - ord('A') + 1 if type(x) == str else 0 for x **in** col])).set\_output(transform="pandas")

extract\_cabin\_named.fit\_transform(train[['Cabin']])

/opt/conda/lib/python3.10/site-packages/sklearn/preprocessing/\_function\_transformer.py:343: UserWarning: With transform="pandas", `func` should return a DataFrame to follow the set\_output API.

warnings.warn(

Out[44]:

|  | Cabin |
| --- | --- |
| PassengerId |  |
| 1 | 0 |
| 2 | 3 |
| 3 | 0 |
| 4 | 3 |
| 5 | 0 |
| ... | ... |
| 887 | 0 |
| 888 | 2 |
| 889 | 0 |
| 890 | 3 |
| 891 | 0 |

891 rows × 1 columns

Now, we can get named output by appending .set\_output(transform='pandas') onto the ColumnTransformer. (The set\_output method looks like it recursively applies itself to subtransformations.)

In [45]:

named\_preprocessing\_pipeline = Pipeline([('extract\_title', ExtractTitle(min\_relative\_freq = 0.2)),

('discretize\_Fare', QCutFare(13)),

('fractional\_Age', AddFractionalAgeColumn()),

('family\_Size', AddSumOfColumnsColumn(columns=['SibSp','Parch'],

constant=1,

name='FamilySize')),

('drop', DropColumn(cols=['Ticket'])),

('prep',ColumnTransformer([('impute',

SimpleImputer(strategy='mean'),

['Age']),

('cabin\_extract',

extract\_cabin\_named,

['Cabin']),

('encode',

OneHotEncoder(handle\_unknown='ignore',

sparse\_output=False),

['Sex', 'Embarked', 'Title']),

],

remainder='passthrough').set\_output(transform='pandas')),

])

named\_preprocessing\_pipeline

/opt/conda/lib/python3.10/site-packages/sklearn/preprocessing/\_function\_transformer.py:343: UserWarning: With transform="pandas", `func` should return a DataFrame to follow the set\_output API.

warnings.warn(

Out[45]:

Pipeline

ExtractTitle

QCutFare

AddFractionalAgeColumn

AddSumOfColumnsColumn

DropColumn

prep: ColumnTransformer

impute

SimpleImputer

cabin\_extract

FunctionTransformer

encode

OneHotEncoder

remainder

passthrough

In [46]:

named\_preprocessing\_pipeline.fit\_transform(train.drop(['Survived'], axis=1))

Out[46]:

|  | impute\_\_Age | cabin\_extract\_\_Cabin | encode\_\_Sex\_female | encode\_\_Sex\_male | encode\_\_Embarked\_C | encode\_\_Embarked\_Q | encode\_\_Embarked\_S | encode\_\_Embarked\_nan | encode\_\_Title\_Miss | encode\_\_Title\_Mr | encode\_\_Title\_Rare/Unknown | remainder\_\_Pclass | remainder\_\_SibSp | remainder\_\_Parch | remainder\_\_QCut13\_Fare | remainder\_\_FractionalAge | remainder\_\_FamilySize |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PassengerId |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 | 22.000000 | 0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3 | 1 | 0 | 1 | 0 | 2 |
| 2 | 38.000000 | 3 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1 | 1 | 0 | 11 | 0 | 2 |
| 3 | 26.000000 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 3 | 0 | 0 | 3 | 0 | 1 |
| 4 | 35.000000 | 3 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 1 | 1 | 0 | 10 | 0 | 2 |
| 5 | 35.000000 | 0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3 | 0 | 0 | 3 | 0 | 1 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 887 | 27.000000 | 0 | 0.0 | 1.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 2 | 0 | 0 | 5 | 0 | 1 |
| 888 | 19.000000 | 2 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 1 | 0 | 0 | 9 | 0 | 1 |
| 889 | 29.699118 | 0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 3 | 1 | 2 | 7 | 0 | 4 |
| 890 | 26.000000 | 3 | 0.0 | 1.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 1 | 0 | 0 | 9 | 0 | 1 |
| 891 | 32.000000 | 0 | 0.0 | 1.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 3 | 0 | 0 | 1 | 0 | 1 |

891 rows × 17 columns

10. Modelling using the pipeline

The final step is attaching a model onto the end of the pipeline. Let's use LogisticRegression again, with a normalization step just before the model:

In [47]:

modelling\_pipeline = Pipeline(named\_preprocessing\_pipeline.steps + \

[('scale',StandardScaler().set\_output(transform='pandas')),

('logreg', LogisticRegression(max\_iter=2500))])

modelling\_pipeline

Out[47]:

Pipeline

ExtractTitle

QCutFare

AddFractionalAgeColumn

AddSumOfColumnsColumn

DropColumn

prep: ColumnTransformer

impute

SimpleImputer

cabin\_extract

FunctionTransformer

encode

OneHotEncoder

remainder

passthrough

StandardScaler

LogisticRegression

In [48]:

modelling\_pipeline.fit(X=train.drop(['Survived'], axis=1),

y=train['Survived'])

predictions = modelling\_pipeline.predict(test)

sample['Survived'] = predictions

sample.to\_csv('accuracy\_submission.csv', index=False)

In [49]:

pipeline\_cv = cross\_validate(estimator = modelling\_pipeline,

X = train.drop(['Survived'], axis=1),

y = train['Survived'],

cv = skf,

scoring = 'accuracy')

print(f"Average cross-validated accuracy from final pipeline: **{**pipeline\_cv['test\_score'].mean()**:**.3f**}**")

Average cross-validated accuracy from final pipeline: 0.808

Let's have a look at feature importances (defined for the logistic regression model as absolute value of the coefficients on scaled input data). Note that we only get column names from the output by using the set\_output method as described in section 9.

In [50]:

plt.figure(figsize=(6,10))

modelling\_pipeline.fit(trainX, trainy)

x\_plt = modelling\_pipeline.steps[-1][1].coef\_.ravel()

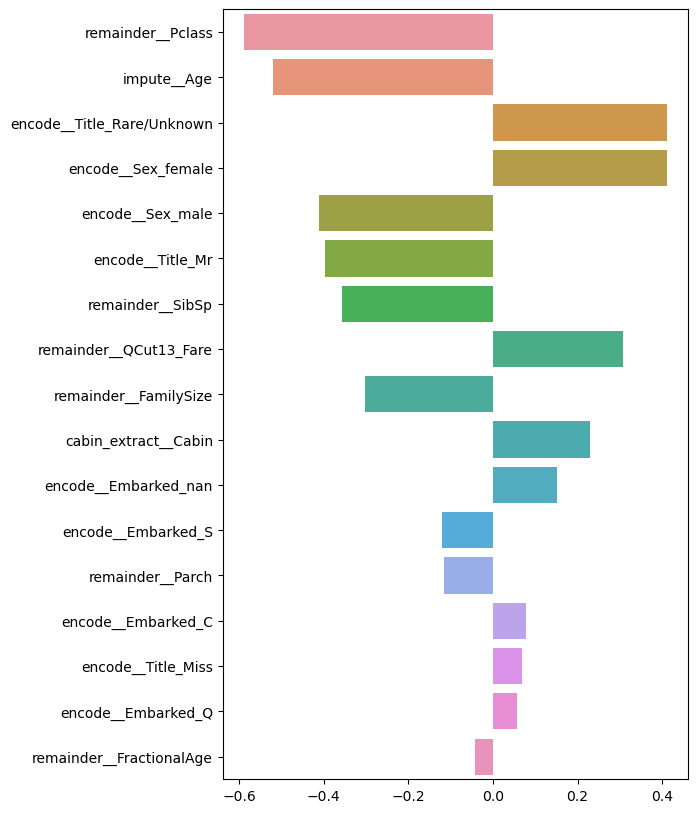
x\_plt\_abs = np.abs(x\_plt)

y\_plt = Pipeline(modelling\_pipeline.steps[:-1]).fit\_transform(trainX).columns

\_,xp,yp=list(zip(\*sorted(list(zip(x\_plt\_abs, x\_plt, y\_plt)),

reverse=True)))

\_=sns.barplot(x=list(xp),y=list(yp))



We see that young, female passengers paying a larger fare are more likely to survive compared to poor, male passengers

**Spoiler Alert:** This is the plot of the movie Titanic

Notice, however, that the 'FractionalAge' feature that we lovingly engineered doesn't look to be particularly useful. Maybe we should get rid of it. One way to drop this preprocessing step without re-writing the pipeline is to use the set\_params() method of the pipeline:

modelling\_pipeline.set\_params(fractional\_Age=None)

But, since everything is in a pipeline, we can explicitly test the effect on the cross-validated score of keeeping or dropping this (or any other) preprocessing step.

11. Testing preprocessing choices using GridSearchCV

One of the great advantages of setting up preprocessing in a pipeline like this is that we can use GridSearchCV to tune the preprocessing choices in conjunction with any model hyperparameters. The [pipeline documentation](https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html) specifies how to do this:

The purpose of the pipeline is to assemble several steps that can be cross-validated together while setting different parameters. For this, it enables setting parameters of the various steps using their names and the parameter name separated by a '\_\_', as in the example below. A step’s estimator may be replaced entirely by setting the parameter with its name to another estimator, or a transformer removed by setting it to 'passthrough' or None.

For this example, I:

1. tune the regularization parameter of the logistic regression, with 'logreg\_\_C' parameter (the 'logreg' is from the name associated with the LogisticRegression step of the pipeline, and the 'C' refers to the logistic regression regularization parameter
2. choose whether the fractional age indicator feature engineering should be included (setting a step to None turns this step off),
3. swap out the mean imputation strategy for a median imputation strategy (notice the chain of parameter naming to access the SimpleImputer strategy here), and
4. Choose the number of bins to discretize 'Fare' with.

We can get a list of the tunable parameters (and their names) by calling get\_params() on the pipeline.

In [51]:

%%time

from sklearn.model\_selection import GridSearchCV

gscv\_roc = GridSearchCV(estimator = modelling\_pipeline,

cv = skf,

scoring='roc\_auc',

verbose=0,

param\_grid = {'logreg\_\_C': [0.001,1,1000],

'fractional\_Age': [None,AddFractionalAgeColumn()],

'prep\_\_impute\_\_strategy': ['mean', 'median'],

'discretize\_Fare\_\_n\_bins': [5,13],

},

)

\_=gscv\_roc.fit(X = train.drop(['Survived'], axis=1),

y = train['Survived'],

)

CPU times: user 15.7 s, sys: 11 s, total: 26.8 s

Wall time: 8.17 s

Note that I changed the scoring metric here to [ROC\_AUC](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.roc_auc_score.html) just to differentiate between the scores a bit better, as accuracy tends to be very similar for the titanic data. We can see the effect of our different preprocessing from the mean test score variable from the GridSearchCV.cv\_results\_ object:

In [52]:

cv\_res = pd.DataFrame({k:[[str(y)[:6] for y **in** x] for x **in** [gscv\_roc.cv\_results\_['param\_'+param\_name].data for \

param\_name **in** gscv\_roc.param\_grid]][i] for \

i,k **in** enumerate(gscv\_roc.param\_grid)})

cv\_res['mean\_cross\_validated\_ROC\_AUC'] = gscv\_roc.cv\_results\_['mean\_test\_score']

cv\_res['rank'] = gscv\_roc.cv\_results\_['rank\_test\_score']

cv\_res.to\_csv('accuracy\_cv.csv', index=False)

cv\_res.sort\_values(by = ['rank'])

Out[52]:

|  | logreg\_\_C | fractional\_Age | prep\_\_impute\_\_strategy | discretize\_Fare\_\_n\_bins | mean\_cross\_validated\_ROC\_AUC | rank |
| --- | --- | --- | --- | --- | --- | --- |
| 2 | 1 | None | mean | 5 | 0.866432 | 1 |
| 8 | 1 | AddFra | mean | 5 | 0.866416 | 2 |
| 10 | 1000 | AddFra | mean | 5 | 0.866192 | 3 |
| 4 | 1000 | None | mean | 5 | 0.866128 | 4 |
| 3 | 1 | None | median | 5 | 0.865920 | 5 |
| 5 | 1000 | None | median | 5 | 0.865809 | 6 |
| 9 | 1 | AddFra | median | 5 | 0.865809 | 6 |
| 11 | 1000 | AddFra | median | 5 | 0.865713 | 8 |
| 15 | 1 | None | median | 13 | 0.864882 | 9 |
| 14 | 1 | None | mean | 13 | 0.864794 | 10 |
| 21 | 1 | AddFra | median | 13 | 0.864786 | 11 |
| 20 | 1 | AddFra | mean | 13 | 0.864698 | 12 |
| 17 | 1000 | None | median | 13 | 0.864562 | 13 |
| 23 | 1000 | AddFra | median | 13 | 0.864498 | 14 |
| 22 | 1000 | AddFra | mean | 13 | 0.864363 | 15 |
| 16 | 1000 | None | mean | 13 | 0.864299 | 16 |
| 18 | 0.001 | AddFra | mean | 13 | 0.856262 | 17 |
| 0 | 0.001 | None | mean | 5 | 0.856254 | 18 |
| 19 | 0.001 | AddFra | median | 13 | 0.856238 | 19 |
| 12 | 0.001 | None | mean | 13 | 0.856230 | 20 |
| 1 | 0.001 | None | median | 5 | 0.856222 | 21 |
| 13 | 0.001 | None | median | 13 | 0.856190 | 22 |
| 7 | 0.001 | AddFra | median | 5 | 0.856142 | 23 |
| 6 | 0.001 | AddFra | mean | 5 | 0.856126 | 24 |

In [53]:

for pname **in** gscv\_roc.param\_grid:

print(f'--- **{**pname**}** ---')

print(cv\_res.groupby(pname)['mean\_cross\_validated\_ROC\_AUC'].mean())

print('')

--- logreg\_\_C ---

logreg\_\_C

0.001 0.856208

1 0.865467

1000 0.865195

Name: mean\_cross\_validated\_ROC\_AUC, dtype: float64

--- fractional\_Age ---

fractional\_Age

AddFra 0.86227

None 0.86231

Name: mean\_cross\_validated\_ROC\_AUC, dtype: float64

--- prep\_\_impute\_\_strategy ---

prep\_\_impute\_\_strategy

mean 0.862349

median 0.862231

Name: mean\_cross\_validated\_ROC\_AUC, dtype: float64

--- discretize\_Fare\_\_n\_bins ---

discretize\_Fare\_\_n\_bins

13 0.861817

5 0.862764

Name: mean\_cross\_validated\_ROC\_AUC, dtype: float64

So, we see that the fractional age indicator variable doesn't help predictions, mean imputation is slightly better than median imputation, and discretizing Fare into 5 bins rather than 13 improves our model.

12. Limitations of pipelines

Although pipelines have a lot of flexibility, there are a few limitations:

1. Preprocessing pipelines can end up being very CPU-intensive. If we are testing a lot of model parameters with exactly the same preprocessing steps, it will probably be worthwhile splittling the pipeline into two, and running the preprocessing pipeline only once. Pipeline has a 'memory' option, which can cache preprocessing steps, but I have not got around to using it properly.
2. The output from pipeline and transformers is an array rather than pandas dataframe. This means that:
   1. we can't chain processors by name in a pipeline. For example, the 'Cabin' feature extraction (section 6) returns an array rather than a named pandas Series. If we wanted to encode this in a separate, downstream, step (for example, along with the 'Sex' and 'Embarked' columns), we would have to refer to the column numbers in the ColumnTransformer step rather than the dataframe column names. ColumnTransformer also reorders columns (returning the transformed columns in order of the transformers and with the pass-through columns at the end), so it is difficult to tell which column number refers to which transformed column.
   2. we can't use column names from the output of a ColumnTransformer or Pipeline e.g. for feature importance. Model inference (e.g. feature importance scores, or even regression parameter standard errors) is hard to do when the column names are lost from the preprocessing.
3. Although pipelines (and transformers) accept the target variable y, which can in theory be used to transform the. It would've been good to have this in a pipeline to test the effect of target imputation, but it appears to be currently impossible, without modifying the pipeline API: the  provides a workaround by over-riding the sklearn Pipeline class, which works quite well, but it's a bit fiddly to use with custom transformers.

13. Conclusions

Pipelines are a way of organising data processing and/or modelling steps into one object. This is a better way of organising your data-science workflow.

There are several benefits of using a pipeline, and these include:

1. Eliminating duplicated code.
2. Avoiding data leakage by processing training and test data separately.
3. Embedding preprocessing in a cross-validation scheme.
4. Code readability.
5. Using pipelines allows the possibility of tuning preprocessing choices for better test predictions.