Overview

In this notebook we will learn about

- Categorical variables
    - non-numeric
- Classification task
    - Supervised Learning, with categorical target

We will begin with the problem of Binary Classification, in which the targets are one of two possible values

- Positive/Negative

Recipe Step A: Get the data

Frame the problem

Borrowed from [Wikipedia (https://en.wikipedia.org/wiki/RMS_Titanic)](https://en.wikipedia.org/wiki/RMS_Titanic)

> *RMS Titanic was a British passenger liner that sank in the North Atlantic Ocean in 1912 after the ship struck an iceberg during her maiden voyage from Southampton to New York City. Of the estimated 2,224 passengers and crew aboard, more than 1,500 died, making it one of modern history's deadliest peacetime commercial marine disasters*

The goal is to predict whether a passenger survives, based on passenger characteristics.

- target: { "Survive", "Not Survive" }
- features: vector of passenger characteristics

Aside: What does the Titanic have to do with Finance or Risk ?

- Defaults (Survival probability of a corporation)
    - Credit risk
- Mortgage pre-payment

Recipe A.1: Get the data

The data comes in two CSV format files

- train
- test

We will read them into a Pandas DataFrame.

- Observe our use of *relative paths* for file names
- Using relative rather than absolute paths is a requirement of your assignments!

In [5]:
```
# Note the use of *relative path*; your assignments should all use relative rath
er than absolute paths
TITANIC_PATH = os.path.join("./external/jack-dies", "data")

train_data = pd.read_csv( os.path.join(TITANIC_PATH, "train.csv") )
test_data  = pd.read_csv( os.path.join(TITANIC_PATH, "test.csv")  )
```

In [6]:

```
train_data.shape
test_data.shape
```

Out[6]:

```
(891, 12)
```

Out[6]:

```
(418, 11)
```

Recipe A.2: Have a look at the data

Let's examine the first few records to get a feel for the shape of the data.

This will help us understand the features and the target.

In [7]:

```
train_data.columns
train_data.head()
```

Out[7]:

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

Out[7]:

	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

The attributes have the following meaning:

- **Survived**: that's the target
  - 0 means passenger did not survive
  - 1 means passenger survived.
- **Pclass**: passenger class.
- **Name**, **Sex**, **Age**: self-explanatory
- **SibSp**: how many siblings & spouses of the passenger aboard the Titanic.
- **Parch**: how many children & parents of the passenger aboard the Titanic.
- **Ticket**: ticket id
- **Fare**: price paid (in pounds)
- **Cabin**: passenger's cabin number
- **Embarked**: where the passenger embarked the Titanic

How many observation are there ?

In [8]:

```
(num_obs, num_features) = train_data.shape
print("There are {nr} observations and {nf} attributes (including the target)".f
ormat(nr=num_obs, nf=num_features) )
```

```
There are 891 observations and 12 attributes (including the target)
```

The first thing to notice is that we have many fewer examples than the number of passengers indicated in the problem statement.

We'll ignore this for now.

Let's try to understand the types of the attributes

In [9]:
```
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Non-numeric attributes

We can see the non-numeric attributes (type is "object") are:

- `Name`
- `Sex`
- `Cabin`
- `Embarked`

Data issues: missing attributes

We can also see that we have some missing data issues to deal with.

Any attribute with less than num_obs values has observations with a missing value in the attribute

- `Age`
- `Cabin`
- `Embarked`

Other issues

- Shouldn't `Survived` be non-numeric (Positve/Negative or Survived/Not) ?
    - looks like this has been encoded as the integer 1/0
- What about `Pclass` ?
    - This could have just as easily been present as non-numeric "First", "Second", "Third"
    - Is the fact that it has been encoded as integers significant ?
    - This is much deeper than it sounds.
        - We will revisit when discussing Categorical variables.

For our first pass at the problem: we will ignore issues concering `Survived` and `Pclass`.

Let's get a summary of the distribution of each attribute

(n.b., `describe` operates *only* on the numeric attributes)

In [10]:

```
train_data.describe()
```

Out[10]:

	PassengerId	Survived	Pclass	Age	SibSp	Parch	Fare
count	891.000000	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	446.000000	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	257.353842	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	1.000000	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	223.500000	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	446.000000	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	668.500000	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	891.000000	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

- You can also observe the attributes with missing values by looking at the "count"
- You can clearly see that `Survived` is a binary, *integer* variable
- Only 38 % of the passengers survived ("mean")

In [11]:

```
train_data["Survived"].value_counts()
```

Out[11]:

```
0    549
1    342
Name: Survived, dtype: int64
```

Return to parent notebook

Recipe A.3: Select a performance measure

Our performance measure will be **accuracy**, the fraction of correct predictions.

$$\text{Accuracy} = \frac{\text{number of correct predictions}}{\text{number of predictions}}$$

There are several drawbacks with this definition, which we will address later.

But let's start with it for now.

Recipe A.4: Create a test set and put it aside !

The train/test split was done for us: it came as two separate files

We might later choose to combine the two and do our own split (or better yet: multiple splits) but for now, we'll take what we are given.

Note that the test set provided *does not have targets* associated with each example

- The dataset was from a Kaggle competition. The "answers" (targets) to the test set were known only to the judges.
- This means you can't use the provided test set to evaluate the Performance Measure. Either
    - Create your own test set
    - Use Cross Validation

Return to parent notebook

Recipe Step B: Exploratory Data Analysis (EDA)

Visualize Data to gain insights

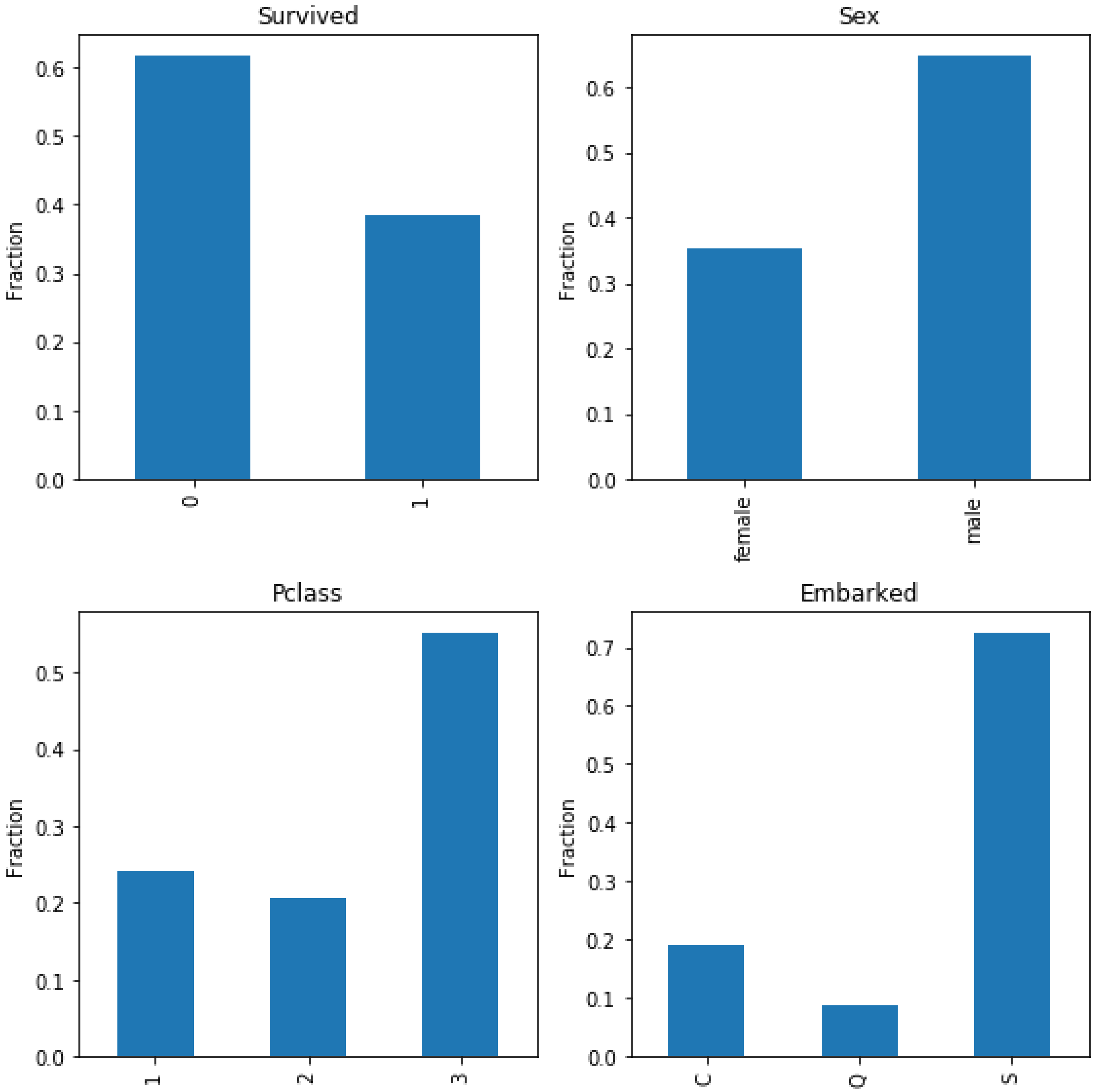
Distribution of each attribute

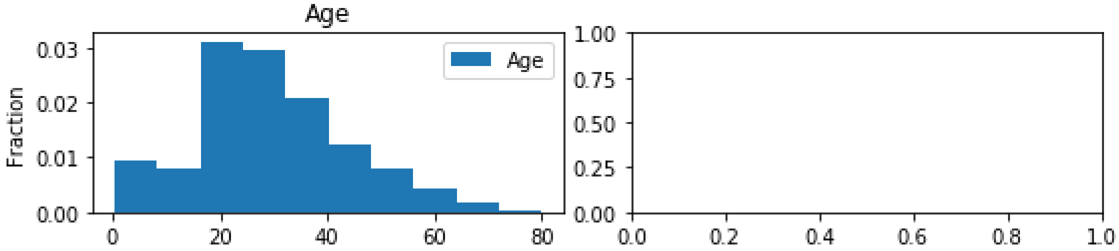
Let's start by looking at the (unconditional) distribution of the target and some attributes

First let's look at them normalized (i.e, as fractions or probabilities)

In [12]:

```
clh.plot_attrs(train_data, [ "Survived", "Sex", "Pclass", "Embarked" ], attr_typ
e="Cat", plot=True, normalize=True)
clh.plot_attrs(train_data, [ "Age" ], attr_type="Num", normalize=True)
```

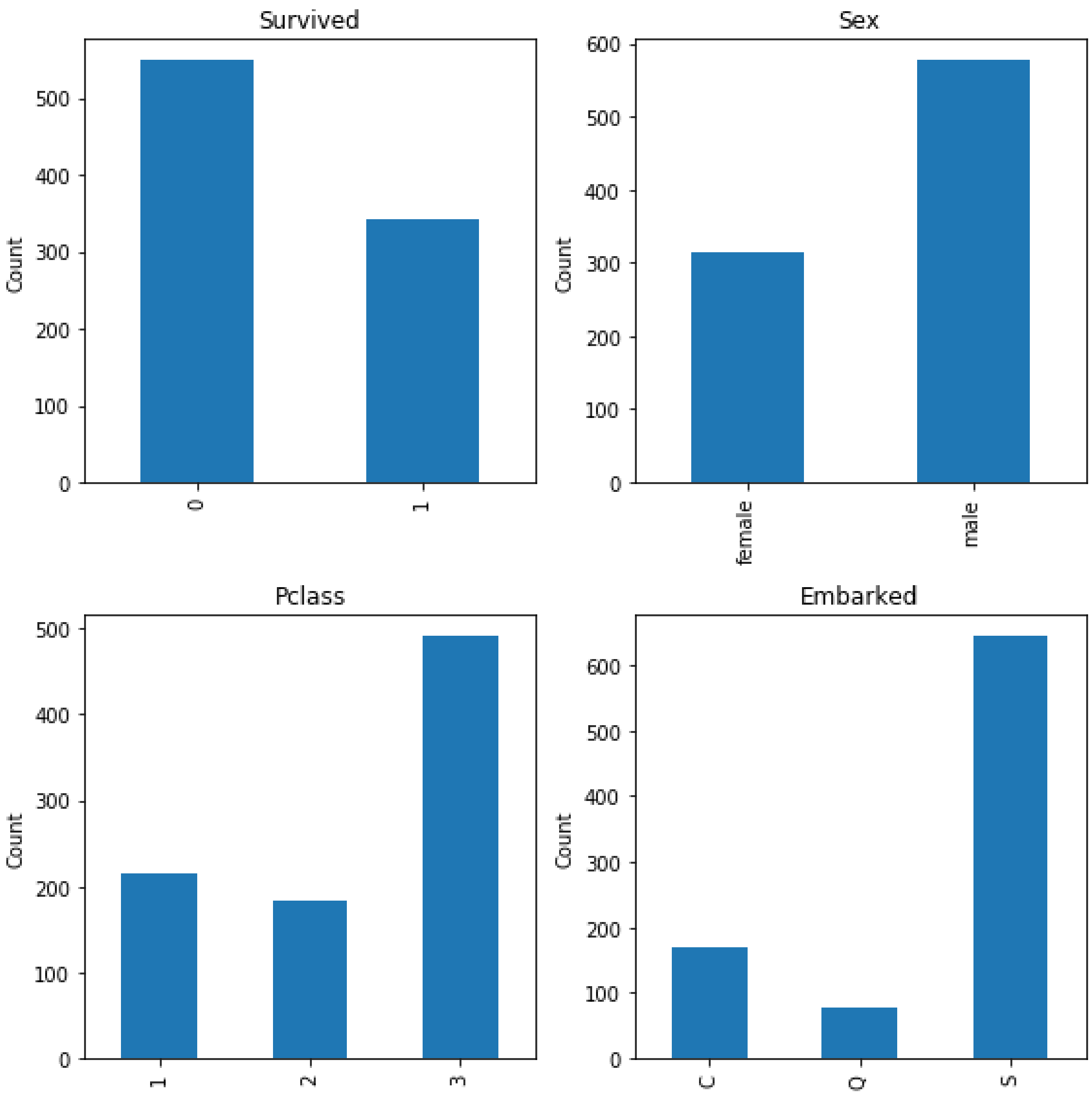


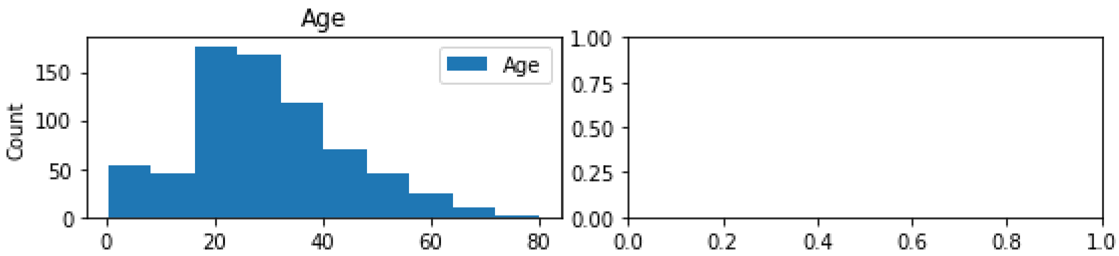

Age
Fraction
Age
0.00
0.01
0.02
0.03
0
20
40
60
80
1.00
0.75
0.50
0.25
0.00
0.0
0.2
0.4
0.6
0.8
1.0

Next, let's look at them un-normalized, or absolute count

In [13]:

```
clh.plot_attrs(train_data, [ "Survived", "Sex", "Pclass", "Embarked" ], attr_typ
e="Cat", plot=True, normalize=False)
clh.plot_attrs(train_data, [ "Age" ], attr_type="Num", normalize=False)
```





Age
Age
Count
150
100
50
0
0
20
40
60
80
1.00
0.75
0.50
0.25
0.00
0.0
0.2
0.4
0.6
0.8
1.0

Conditional survival probability (condition on single attribute)

Let's explore whether there is a relationship between the target (Survive) and single features

- This might tell us whether the feature has some predictive value
    - If the distribution of the target is different for each value of the feature
    - The feature may have an association with the target

In [14]:

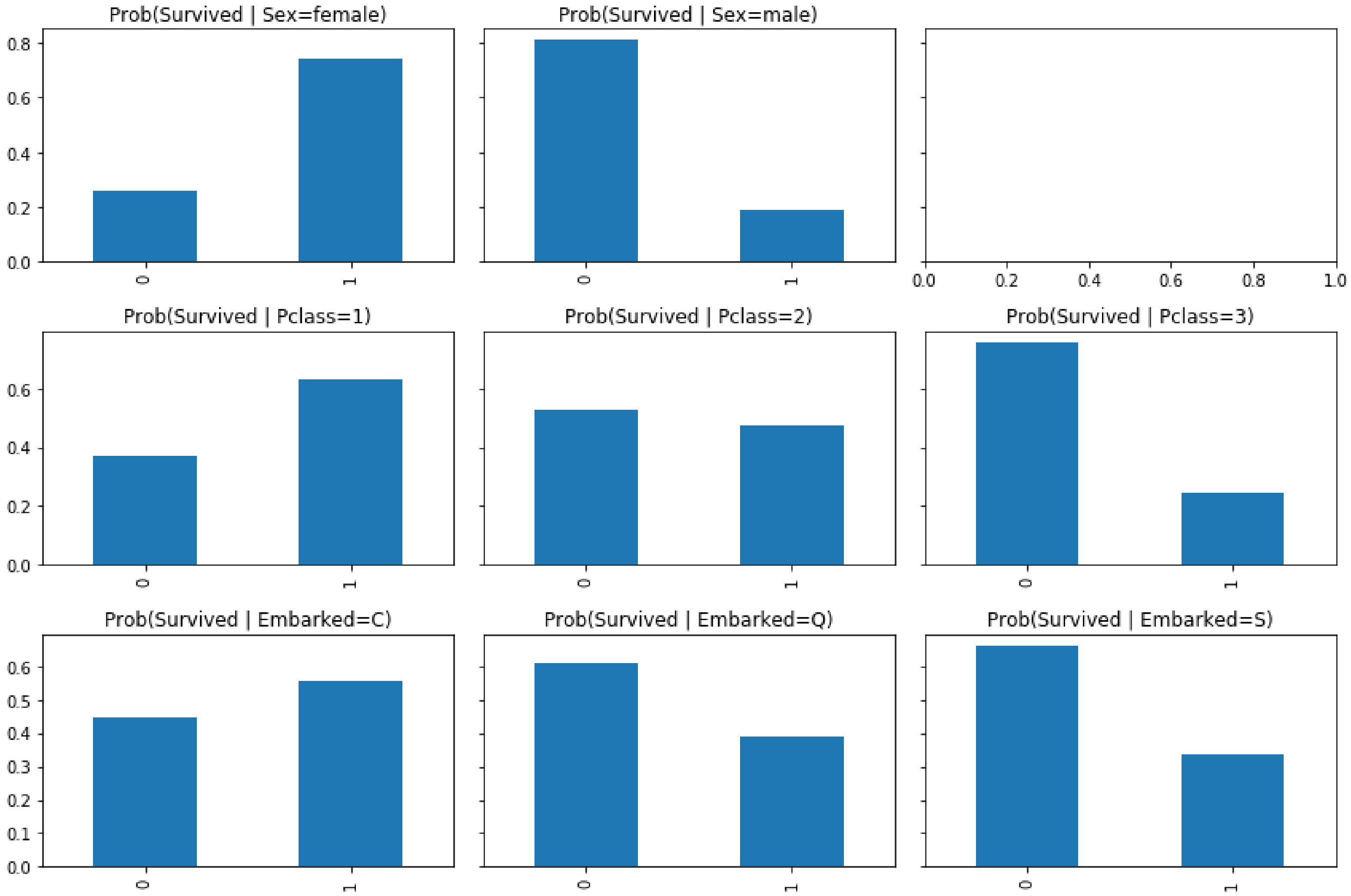
```
fig, axs = clh.plot_conds(train_data, [ ("Survived", { "Sex": "female"}),
                         ("Survived", { "Sex": "male"}),
                         None,
                         ("Survived", { "Pclass": 1}),
                         ("Survived", { "Pclass": 2}),
                         ("Survived", { "Pclass": 3}),

                         ("Survived", { "Embarked":"C"}),
                         ("Survived", { "Embarked":"Q"}),
                         ("Survived", { "Embarked":"S"})
                       ],
                          share_y="row",
                          normalize="False"
          )

# In case the figure is truncated in this slide, we will re-display it in the ne
xt (without the code)
plt.close(fig)
```

In [15]:
```
# Display the figure again, in it's own slide, so it is not truncated
fig
```

Out[15]:



Interesting !

- Women are 3 times as likely to survive
- NOT being in the lowest Class doubles or triples your survival probability
- Embarking at Cherbourg increased your probability of surviving
    - WHY ? Is there a correlation between Class and point of embarcation maybe ?

Preview: There may also be lessons here for dealing with missing data

Conditional survival probability (condition on multiple attributes)

Just as we examined possible associations between the target and single features, we now examine a possible association to pairs of features

- A single feature alone may not separate the data into different target values
- But combinations of features might

In [16]:

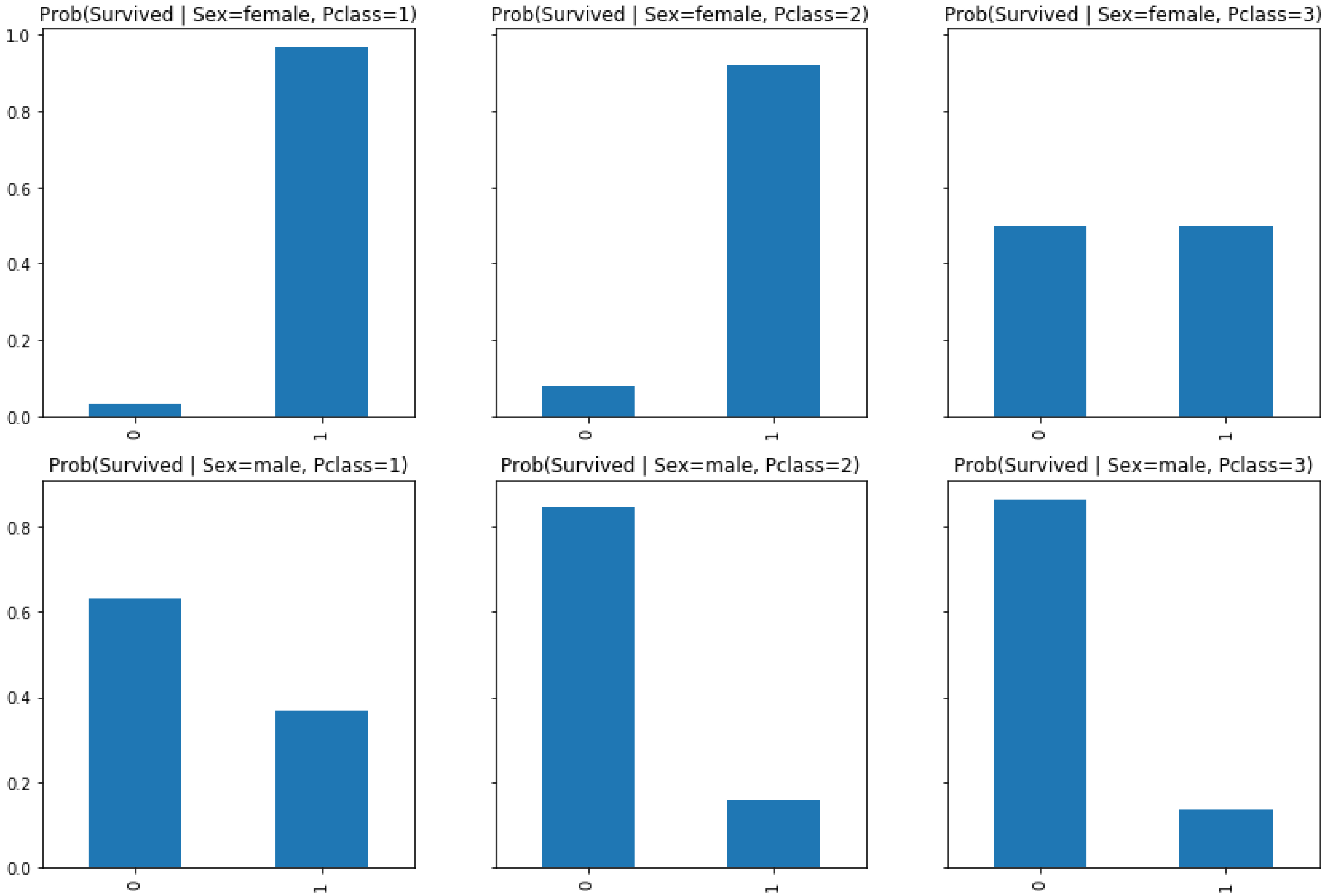
```
fig, axs = clh.plot_conds(train_data, [
                         ("Survived", { "Sex": "female", "Pclass": 1}),
                         ("Survived", { "Sex": "female", "Pclass": 2}),
                         ("Survived", { "Sex": "female", "Pclass": 3}),

                         ("Survived", { "Sex": "male", "Pclass": 1}),
                         ("Survived", { "Sex": "male", "Pclass": 2}),
                         ("Survived", { "Sex": "male", "Pclass": 3}),
                       ],
                          share_y="row"
                         )

# In case the figure is truncated in this slide, we will re-display it in the ne
xt (without the code)
plt.close(fig)
```

In [17]:
```python
# Display the figure again, in it's own slide, so it is not truncated
fig
```

Out[17]:



Remember that we thought being Female increased chances of Survive ?

- we can see now see that this was only true for Classes 1 and 2

Aside: Using Pandas for partitioning the data

Aside: How does pd.groupby() work ?

If you examine our modules to see how we are partitioning the data

- You will see how useful the Pandas `groupby` method is
- For those who know SQL: this is similar to grouping in database queries
- For those who want to know how `groupby` works, this section explains
    - Feel free to skip it `

The Pandas Split-Apply-Combine pattern is very powerful

- This is very SQL-like, for those who have used databases
- Below is some simpler Pandas code to show exactly how it works

In [18]:

```
# Partition by the attribute "Sex"
males   = train_data[ train_data["Sex"] == "male"]
females = train_data[ train_data["Sex"] == "female"]

# Aggregate within each group: count,mean.  n.b., only doing this for the "Survi
ved" column
count_males, count_females = males.shape[0], females.shape[0]
survival_males, survival_females = males["Survived"].mean(), females["Survived"]
.mean()

print( "male\t{c}\t{m:0.4f}".format(c=count_males, m=survival_males) )
print( "female\t{c}\t{m:0.4f}".format(c=count_females, m=survival_females) )

# Or, use the pd.groupby
train_data.groupby("Sex").agg(["mean", "count"])["Survived"]
```

```
male    577     0.1889
female  314     0.7420
```

Out[18]:

	mean	count
Sex		
female	0.742038	314
male	0.188908	577

Return to parent notebook

Recipe Step C: Prepare the data

Our first model will use the following features

- Pclass
- Sex
- Age
- SibSp: passenger's number of "same-level" relatives (Sibling, Spouse)
- Parch: passenger's number of "different-level" relatives (Parent, Child)
- Fare

We will follow all the steps for the Prepare the Data step per our Recipe.

But

- We will initially only discuss what we intend to do for each sub-step
- The actual code will be deferred to the end of this section
- We will use a single Pipeline to implement all the steps of Prepare the Data

Recipe C.1: Cleaning

Our initial data exploration revealed some attributes with missing data

- Age
- Cabin
- Embarked

We will address various strategies for dealing with missing data in the module on Data Transformations.

For now, we will take a very simple (and naive) approach

- for numeric attributes: use the median value
- for non-numeric attributes: use the most frequent value

Recipe C.2: Handling non-numeric features/targets

The first step will be to transform the data into a usable form.

We will do minimal transformation for now (more to come in the module on Data Transformations)

- clean the data: deal with missing values
- convert the categorical, binary attribute "Sex" to a number: 0:male, 1: female

(We include Data Cleaning as a transformation, rather than treating it as a separate step)

This is also very naive (and, technically wrong ! Ask me later !) but it will allow us to make our key points with minimum distraction.

We will followup with an in-depth discussion of the *proper* way to handle categorical variables.

Categorical Transformation: Binary variable special case

We will need to transform our categorical features into numbers.

The way to treat non-numeric values is not obvious and there are many *wrong* ways to do it.

Fortunately, binary non-numerical data is an an easy special case (0 or 1 values)

We will initially limit ourselves to binary targets and features.

- This will allow us to focus on the Classification task without distraction.
- We will make a second pass to generalize to categorical variables with more than 2 classes.

Which variables are categorical ?

We can (and should) apply *human logic* to discern which variables are categorical.

But a little code can help:

In [19]:

```
train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

- All of the columns described as `object` are non-numerical and hence likely suspects
    - `Name`, `Sex`, `Cabin`, `Embarked`
- But, as observed during our first look, some numeric columns are *also* candidates
    - `Survived`
    - `PassengerId`, `Pclass`
        - Key determinant: is there an ordering relationship ?
- In particular, our target `Survived` which should be categorical, has beeen encoded as an integer {0,1}.
- So, by luck (or bad encoding) we don't have to convert the categorical Survived feature to a binary digit.
- We will have to convert `Sex` (this is the only categorial feature we retain for our initial model)

Recipe C.3: Transformations

Preview of coming attractions (subsequent lecture)

We won't perform feature engineering now other than to point out some interesting possibilities.

This may help you as you tackle problems between now and the lecture on Transformations.

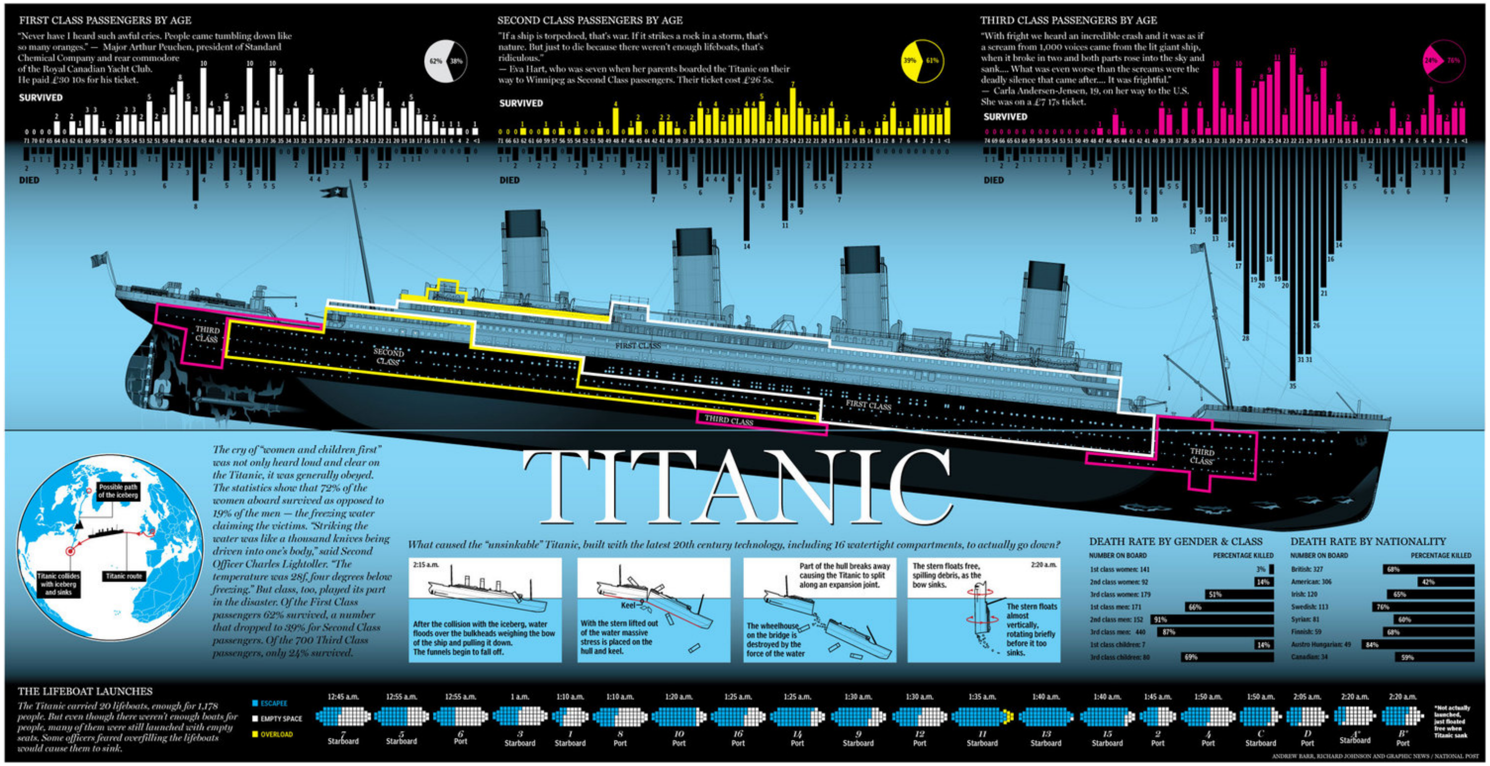
Cabin

Might the Cabin location be associated with Surivival ?

Note that the front of the ship went down first.

If you are a diligent Data Scientist you can find this image, which is revealing

- Worst cabins (pink, Third class) were dispersed between front and back
- Best cabins (white, First class) were dispersed between above/below deck
- Mid cabins(yellow, Second Class) were *not* near front of ship



Age

For linear estimators ($\Theta^T \mathbf{x}$) each increment in value of a feature impacts the prediction.

So an example with feature value $2 * \mathbf{x}_j$ has twice the impact of an example with feature value $\mathbf{x}_j$.

But is this really true ? Does even a small increment in Age (e.g, from 25 to 26 years) matter ?

We might be able to improve things using a "bucket" transformation:

- all Ages within a range (bucket) are given the same value
- model will try to make distinctions *across* buckets, but not *within* a bucket.

In [20]:

```
train_data_augmented = train_data.copy()
train_data_augmented["AgeBucket"] = train_data["Age"] // 15 * 15
train_data_augmented[["AgeBucket", "Survived"]].groupby(['AgeBucket']).mean()
```

Out[20]:

AgeBucket	Survived
0.0	0.576923
15.0	0.362745
30.0	0.423256
45.0	0.404494
60.0	0.240000
75.0	1.000000

Wow ! Children below the age of 15 (bucket 0.0) had a much better chance of survival.

(And it doesn't pay to be old when disaster strikes!)

We would be hard pressed to see this using Age as a continous variable

Embarked

The Embarked attribute tells us where the passenger boarded:

- C=Cherbourg, Q=Queenstown, S=Southampton.

Could this be a predictor of Survival ?

Recipe C.4: Scaling

Nothing to do heere

Recipe Step C using a sophisticated pipeline

We are using an `sklearn` Pipeline to implement the steps to Prepare the Data.

We introduced the tools in [Transformation pipelines in `sklearn` (Transformations_Pipelines.ipynb)](Transformations_Pipelines.ipynb).

We will begin by

- eliminating training examples that don't have targets
- removing the target column from the training data
    - To be sure we don't accidentally cheat
    - The test data may not have a target
        - want number of features in train/test to be identical when performing transformations

In [21]:

```
# Remove examples where target is not defined
train_data = train_data[ train_data["Survived"].notnull() ]

# Separate target from features
y_train = train_data["Survived"]
train_data.drop(columns=["Survived"], inplace=True)
```

The numeric pipeline

- select DataFrame columns identified as Numeric (variable `num_features`)
- perform a "missing value" transformation
    - replace missing value with median value of the feature

In [22]:

```
num_features = ["Age", "SibSp", "Parch", "Fare", "Pclass"]
num_transformers= Pipeline(steps=[  ('imputer', SimpleImputer(strategy='median'
)) ] )
num_pipeline = ColumnTransformer( transformers=[ ("numeric", num_transformers, n
um_features) ] )
```

Let's see what the numeric pipleine produces.

Note: it produces a NumPy array because the final transformer (`SimpleImputer`) does it

In [23]:
```
num_pipeline.fit_transform(train_data)[:num_head]
```

Out[23]:
```
array([[22.    ,  1.    ,  0.    ,  7.25  ,  3.    ],
       [38.    ,  1.    ,  0.    , 71.2833,  1.    ],
       [26.    ,  0.    ,  0.    ,  7.925 ,  3.    ],
       [35.    ,  1.    ,  0.    , 53.1   ,  1.    ],
       [35.    ,  0.    ,  0.    ,  8.05  ,  3.    ]])
```

The non-numeric pipeline

- select DataFrame columns identified as Categorical (variable `cat_features`)
- perform a "missing value" transformation
    - replace missing value with the value of the feature that occurs most often
- perform a "to categorical" transformation on `Sex`

WARNING: I'm doing a little cheating in this code by ignoring all categorical features other than `Sex`

To show the power of `sklearn` transformations

- We will define our own transformations rather than using the one's from `sklearn`
- Doing so allows all of our transformations to consume and produce Pandas DataFrame objects
    - `sklearn` built-in transformations consume and produce NumPy arrays
    - We prefer to refer to features by names, rather than column indices, hence we use Pandas

The code to use the built-in transformation by missing data imputation would be

```
SimpleImputer(strategy="most_frequent")
```

In [24]:

```
cat_features = ["Sex" ]
cat_transformers= Pipeline(steps=[  ('imputer', SimpleImputer(strategy="most_frequent")),
                                    ('sex_encoder', OrdinalEncoder())
                                  ] )
cat_pipeline = ColumnTransformer( transformers=[ ("categorical", cat_transformers, cat_features) ] )
```

In [25]:

```
cat_pipeline.fit_transform(train_data)[:num_head]
train_data["Sex"].head(num_head)
```

Out[25]:

```
array([[1.],
       [0.],
       [0.],
       [0.],
       [1.]])
```

Out[25]:

```
0      male
1    female
2    female
3    female
4      male
Name: Sex, dtype: object
```

Combining the numeric and categorical pipelines: `ColumnTransformer`

The "official" way to combine pipelines in `sklearn` is via the `FeatureUnion`

- You must manually select the features of each type
- Apply the corresponding Pipelines
- "Glue together" (horizontally) the results of the Pipelines

We will use the experimental `ColumnTransformer`, which combines all these steps.

In [26]:
```
preprocess_pipeline = ColumnTransformer(
    transformers=[ ("numeric", num_transformers, num_features),
                   ("categorical", cat_transformers, cat_features)
                 ]
)
```

Let's look at the result of applying the `preprocess_pipeline`

In [27]:
```
X_train = preprocess_pipeline.fit_transform(train_data)

X_train.shape
X_train[:num_head]

# X_train is now an ndarray, so really can't discern columns, but are in same or
der as in Feature Union
# so first the num_features, then cat_features
# Can verify this by looking at train_data
all_features = num_features.copy()
all_features.extend(cat_features)
train_data.loc[:, all_features ] .head()
```

Out[27]:
```
(891, 6)
```

Out[27]:
```
array([[22.    ,  1.    ,  0.    ,  7.25  ,  3.    ,  1.    ],
       [38.    ,  1.    ,  0.    , 71.2833,  1.    ,  0.    ],
       [26.    ,  0.    ,  0.    ,  7.925 ,  3.    ,  0.    ],
       [35.    ,  1.    ,  0.    , 53.1   ,  1.    ,  0.    ],
       [35.    ,  0.    ,  0.    ,  8.05  ,  3.    ,  1.    ]])
```

Out[27]:

	Age	SibSp	Parch	Fare	Pclass	Sex
0	22.0	1	0	7.2500	3	male
1	38.0	1	0	71.2833	1	female
2	26.0	0	0	7.9250	3	female
3	35.0	1	0	53.1000	1	female
4	35.0	0	0	8.0500	3	male

Return to parent notebook

Recipe Step D: Train a model

OK, we have identified features and now want to predict Survival.

How do we do it ?

Recipe D.1:Select a model

We currently know of two models for the Classification task

- K Nearest Neighbors
- Logistic Regression

We will use Logistic Regression.

To review:

$$\begin{aligned} s &= \Theta^T \mathbf{x} \\ \hat{p} &= \sigma(s) \end{aligned}$$

$$\hat{y}^{(\mathbf{i})} = \begin{cases} \text{Negative} & \text{if } \hat{p}^{(\mathbf{i})} < 0.5 \\ \text{Positive} & \text{if } \hat{p}^{(\mathbf{i})} \geq 0.5 \end{cases}$$

We will re-visit the choice of threshold 0.5 at a later time.

Varying the threshold is an attempt at balancing the impact of making two different types of incorrect prediction

- False Positive: predicting Positive when the true target is Negative
- False Negative: predicting Negative when the true target is Positive

Depending on your problem, the impact of being incorrect is not the same for both cases.

Step D.2: Fit

Before fitting, let's

- Drop any examples for which the target/label "Survived" is not defined
- Remove the target/label "Survived" from the features $\mathbf{X}$; move it to target $\mathbf{y}$

Logistic Regression classifier

Let's instantiate a `LogisticRegression` classifier

In [28]:

```
# New version of sklearn will give a warning if you don't specify a solver (b/c
 the default solver -- liblinear -- will be replaced in future)
logistic_clf = linear_model.LogisticRegression(solver='liblinear')
```

More models, more fun ! Same price !

Although we have selected Logistic Regression as our model,there are other models for the Classification task.

It turns out to be just as easy to run *multiple* models as it is one !

So, for pedantic purposes, we will fit *many* models

- to show you how easy it is
- we will not delve deeply into the other models, at least for now

SVM Classifier

In [29]:

```
from sklearn.svm import SVC

svm_clf = SVC(gamma="auto")
```

Random Forest Classifier

In [30]:

```
from sklearn.ensemble import RandomForestClassifier

forest_clf = RandomForestClassifier(n_estimators=100, random_state=42)
```

Fit the models

We will train all the models at once.

This demonstrates the power of a consistent API:

- training several models is no more difficult than training one.

Cheating in Cross Validation

Here is a common way of cheating

- performing all transformations *before* cross validation

In [31]:

```
for name, clf in { "Logistic": logistic_clf,
                   "SVM": svm_clf,
                   "Random Forest": forest_clf
                 }.items():

    # Transform the data
    X_train = preprocess_pipeline.fit_transform(train_data)
    _= clf.fit(X_train, y_train)

    # Cross validation
    scores = cross_val_score(clf, X_train, y_train, cv=10)
    print("Model: {m:s} avg cross val score={s:3.2f}\n".format(m=name, s=scores.
mean()) )
```

```
Model: Logistic avg cross val score=0.80

Model: SVM avg cross val score=0.71

Model: Random Forest avg cross val score=0.82
```

This is "cheating" because

- The transformation of raw features (`train_data`) into synthetic features (`X_train`)
- Used the *entire* set of training examples
- Even though each iteration of Cross Validation *did not* have one fold as part of its training examples

It might be too strong to call this "cheating" but at the least we are peeking at out of sample examples.

Why is this so common ?

- It seems like a lot of coding effort to run the transformations for each iteration of Cross Validation
- It *is* cheaper to transform the raw data once, rather than k times: one for each fold
- The difference between the proper and improper ways is often not significant

Fortunately, `sklearn` makes it easy to perform Cross Validation without peeking.

(Other Machine Learning toolkits may not make it as easy).

- Create a Pipeline with a classifier as the final element
- Use that Pipeline as the "model" for Cross Validation

In [32]:
```
for name, clf in { "Logistic": logistic_clf,
                   "SVM": svm_clf,
                   "Random Forest": forest_clf
                 }.items():

    # Combine the transformation pipeline with a final classification step
    model_pipeline = Pipeline(steps=[ ("transform", preprocess_pipeline),
                                      ("classify", clf)
                                    ]
                             )

    # Cross validation on the combined pipeline
    scores = cross_val_score(model_pipeline, train_data, y_train, cv=10)
    print("Model: {m:s} avg cross val score={s:3.2f}\n".format(m=name, s=scores.
mean()) )
```

```
Model: Logistic avg cross val score=0.80

Model: SVM avg cross val score=0.71

Model: Random Forest avg cross val score=0.82
```

The (average) accuracy of the various models is in the range of 70% to 80%.

The `model_pipeline` gives us a way to consistenly apply transformations to all examples

- including test examples: `model_pipeline.predict(X_test)`

Nested Pipelines

Notice the use of nested Pipelines:

- `model_pipeline` contains Pipeline `preprocess_pipeline` as an element
- `preprocess_pipeline` contains Pipelines `num_transformers` and `cat_transformers` as elements

Return to parent notebook

Recipe D.4: Error analysis

Cross validation gives you a metric, *at an aggregate level*, of how well the model performed out of sample.

You will gain deeper insight into the Classification task by analyzing *individual* predictions.

You can do this by

- going through each out of sample example
- determine whether the prediction is correct for the example

Is there some class for which the predictions are much less succesful than others ?

This is a signal to improve your model

- perhaps by adding features that aid prediction for the trouble classes.

There is some standard terminology for analyzing classification predictions.

For binary classification, to be concrete, let's call the two classes Positive (P) and Negative (N).

Let's create a table

- the row labels correspond to the predicted class
- the column labels correspond to the target (actual) class

In pictures:

$$
\begin{array}{lll}
 & \mathbf{P} & \mathbf{N} \\
\mathbf{P} & \text{TP} & \text{FP} \\
\mathbf{N} & \text{FN} & \text{TN} \\
\end{array}
$$

- The entry labelled True Positive (TP) denotes the number of test examples
  - that were correctly predicted as Positive
- The entry labelled as False Positive (FP) denotes the number of test examples
  - that were incorrectly predicted as Positive
- The entry labelled True Negative (TN) denotes the number of test examples
  - that were correctly predicted as Negative
- The entry labelled False Negative denotes the number of test examples
  - that were incorrectly predicted as Negative

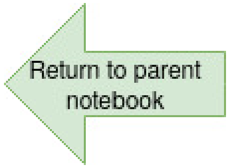
So

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

The False Positive (FP) and False Negative (FN) are our errors.

If the incorrectly predicted examples share something in common

- this may indicate a problem in our model
- needs to be corrected, perhaps through feature engineering.



Categorical data

A categorical variable

- has a finite number of discrete values
    - call each value a *category* or *class* (e.g., classification)
- There is *no* ordering relationship between categories
    - { "Red", "Green", "Blue" } (set notation)
    - versus ["Small", "Medium", "Large"] (sequence notation)
        - *ordinal* if there is an ordering relationship, even in the absence of a magnitude

The Classification task has a categorical target

Titanic revisited: OHE features

Is `Pclass` really a numeric feature just because it was presented to us as a value in $\{1,2,3\}$?

We argue that it should be treated as a *categorical* feature

- Any integer ordering we could impose would also impose a magntiude that could affect the math
    - $\{1,2,3\}$ versus $\{10,20,30\}$

The proper way to deal with categorical variables is with One Hot Encoding.

Let's treat `Pclass` as categorical and refit our model.

- Move `Pclass` from `num_features` to `cat_features`

In [33]:

```
print("First model\n\tcategorical features are: {c:s}\n\tnumeric features are: {n:s}".format(
    c=", ".join(cat_features), n=", ".join(num_features))
    )
```

```
First model
	categorical features are: Sex
	numeric features are: Age, SibSp, Parch, Fare, Pclass
```

In [34]:

```
num_features = ["Age", "SibSp", "Parch", "Fare" ]
num_transformers= Pipeline(steps=[  ('imputer', SimpleImputer(strategy='median'
)) ] )
num_pipeline = ColumnTransformer( transformers=[ ("numeric", num_transformers, n
um_features) ] )

cat_features = ["Sex", "Pclass" ]
cat_transformers= Pipeline(steps=[  ('imputer', SimpleImputer(strategy="most_fre
quent")),
                                    ('cat_encoder', OneHotEncoder(sparse=False))
                                  ] )
cat_pipeline = ColumnTransformer( transformers=[ ("categorical", cat_transformer
s, cat_features) ] )

preprocess_pipeline = ColumnTransformer(
    transformers=[ ("numeric", num_transformers, num_features),
                   ("categorical", cat_transformers, cat_features)
                 ]
)
```

In [35]:

```
print("Second model\n\tcategorical features are: {c:s}\n\tnumeric features are: {n:s}".format(
    c=", ".join(cat_features), n=", ".join(num_features))
    )
```

```
Second model
	categorical features are: Sex, Pclass
	numeric features are: Age, SibSp, Parch, Fare
```

The `cat_pipeline` now creates 5 indicators whereas the old one created only 1

- The `Sex`attribute is now 2 indicators ("Is Female", "Is Male")
- The `Pclass` attribute is now 3 indicators ("Is class 1", "Is class 2", "Is class 3")

Let's look at the first couple of training examples after OHE has been applied.

We will show the encoding along with the corresponding class labels.

In [36]:

```python
# Run the categorial pipeline
cat_ndarray = cat_pipeline.fit_transform(train_data)

# Let's examine the first first rows of the ndarray, and relate them to the same
rows in the DataFrame,
# -- n.b., with the DataFrame, we can see the column names
num_to_see = 7

print(cat_features[0] + ":\n")
cat_ndarray[:num_to_see, :2]

train_data.loc[:, [ cat_features[0] ] ].head(num_to_see)
```

```
Sex:
```

Out[36]:

```
array([[0., 1.],
       [1., 0.],
       [1., 0.],
       [1., 0.],
       [0., 1.],
       [0., 1.],
       [0., 1.]])
```

Out[36]:

	Sex
0	male
1	female
2	female
3	female
4	male
5	male
6	male

In [37]:
```
print(cat_features[1] + ":\n")
cat_ndarray[:num_to_see, -3:]
train_data.loc[:, [ cat_features[1]]].head(num_to_see)
```

```
Pclass:
```

Out[37]:
```
array([[0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [1., 0., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [1., 0., 0.]])
```

Out[37]:

	Pclass
0	3
1	1
2	3
3	1
4	3
5	3
6	1

Notice that $\mathbf{X}$ now has many more feature columns.

In [38]:

```
X_train = preprocess_pipeline.fit_transform(train_data)

# Note: All the columns in untransformed train_data are NOT in X_train, especial
ly the target Survived !
train_data.columns
X_train.shape
X_train
```

Out[38]:

```
Index(['PassengerId', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp', 'Parch',
       'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

Out[38]:

```
(891, 9)
```

Out[38]:

```
array([[22.,  1.,  0., ...,  0.,  0.,  1.],
       [38.,  1.,  0., ...,  1.,  0.,  0.],
       [26.,  0.,  0., ...,  0.,  0.,  1.],
       ...,
       [28.,  1.,  2., ...,  0.,  0.,  1.],
       [26.,  0.,  0., ...,  1.,  0.,  0.],
       [32.,  0.,  0., ...,  0.,  0.,  1.]])
```

What effect did the change in treatment of `Pclass` have ?

In [39]:

```
for name, clf in { "Logistic": logistic_clf,
                   "SVM": svm_clf,
                   "Random Forest": forest_clf
                 }.items():

    model_pipeline = Pipeline(steps=[ ("transform", preprocess_pipeline),
                                      ("classify", clf)
                                    ]
                             )

    # Cross validation
    scores = cross_val_score(model_pipeline, train_data, y_train, cv=10)
    print("Model: {m:s} avg cross val score={s:3.2f}\n".format(m=name, s=scores.
mean()) )
```

```
Model: Logistic avg cross val score=0.79

Model: SVM avg cross val score=0.73

Model: Random Forest avg cross val score=0.81
```

Not too different across models (hard to even know whether the difference is statistically significant).

- Some models (e.g., Random Forest) are *not sensitive* to encoding of categorical
- Logistic Regression *is* potentially sensitive, but it depends on the data
    - would encoding `Pclass` with $\{1000, 2000, 3000\}$ have affected it ?

In [40]:

```
print("Done")
```

Done