**Logistic Regression**

**Feature Selection**

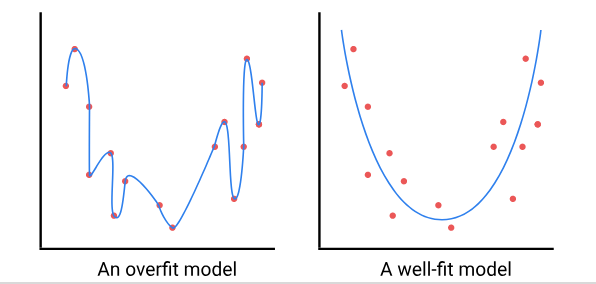
In the last mission, we made our first submission to Kaggle, getting an accuracy score of 75.6%. While this is a good start, there is definitely room for improvement. There are two main areas we can focus on to boost the accuracy of our predictions:

* Improving the features we train our model on
* Improving the model itself

In this mission, we're going to focus working with the features used in our model.

We'll start by looking at **feature selection**. Feature selection is important because it helps to exclude features which are not good predictors, or features that are closely related to each other. Both of these will cause our model to be less accurate, particularly on previously unseen data.

The diagram below illustrates this. The red dots represent the data we are trying to predict, and each of the blue lines represents a different model.

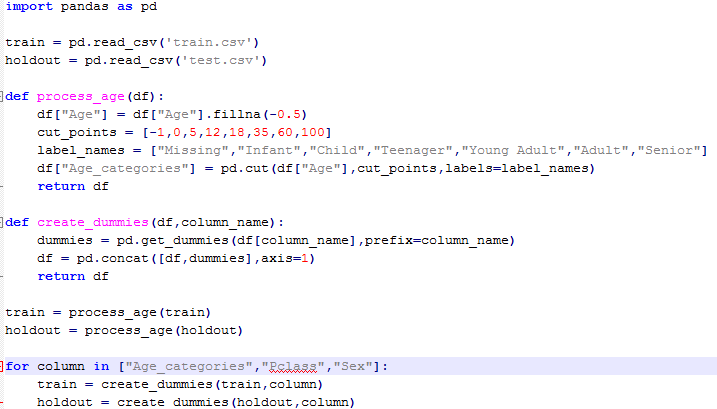


The model on the left is **overfitting**, which means the model represents the training data too closely, and is unlikely to predict well on unseen data, like the holdout data for our Kaggle competition.

The model on the right is **well-fit**. It captures the underlying pattern in the data without the detailed noise found just in the training set. A well fit model is likely to make accurate predictions on previously unseen data. The key to creating a well-fit model is to select the right balance of features, and to create new features to train your model.

In the previous mission, we trained our model using data about the age, sex and class of the passengers on the Titanic. Let's start by using the functions we created in that mission to add the columns we had at the end of the first mission.

Remember that any modifications we make to our training data (train.csv) we also have to make to our holdout data (test.csv).



In order to select the best-performing features, we need a way to measure which of our features are relevant to our outcome - in this case, the survival of each passenger. One effective way is by training a logistic regression model using all of our features, and then looking at the coefficients of each feature.

The scikit-learn [LogisticRegression class](http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html" \t "_blank) has an attribute in which coefficients are stored after the model is fit, LogisticRegression.coef\_. We first need to train our model, after which we can access this attribute.



lr = LogisticRegression()

lr.fit(train\_X,train\_y)

coefficients = lr.coef\_

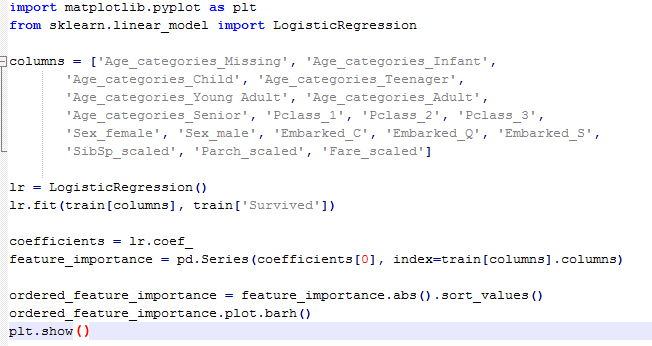
The coef() method returns a NumPy array of coefficients, in the same order as the features that were used to fit the model. To make these easier to interpret, we can convert the coefficients to a pandas series, adding the column names as the index:



feature\_importance = pd.Series(coefficients[0],

                              index=train\_X.columns)

We'll now fit a model and plot the coefficients for each feature.



The plot we generated in the last screen showed a range of both positive and negative values. Whether the value is positive or negative isn't as important in this case, relative to the magnitude of the value. If you think about it, this makes sense. A feature that indicates strongly whether a passenger died is just as useful as a feature that indicates strongly that a passenger survived, given they are mutually exclusive outcomes.

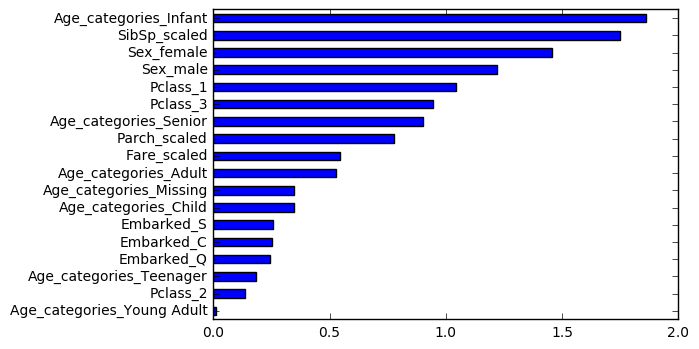
To make things easier to interpret, we'll alter the plot to show all positive values, and have sorted the bars in order of size:



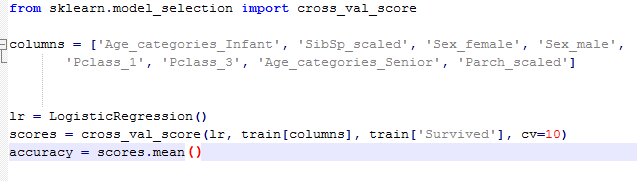
ordered\_feature\_importance = feature\_importance.abs().sort\_values()

ordered\_feature\_importance.plot.barh()

plt.show()

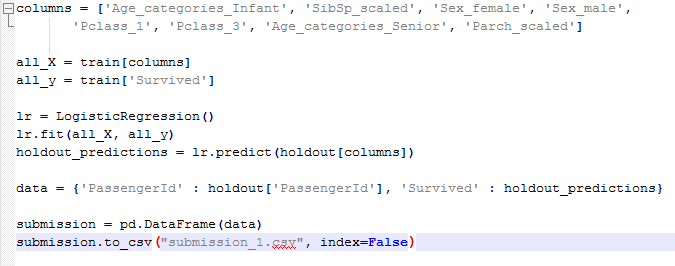


We'll train a new model with the top 8 scores and check our accuracy using cross validation.



The cross validation score of 81.48% is marginally higher than the cross validation score for the model we created in the previous mission, which had a score of 80.2%.

Hopefully, this improvement will translate to previously unseen data. Let's train a model using the columns from the previous step, make some predictions on the holdout data and submit it to Kaggle for scoring



**Feature Engineering**

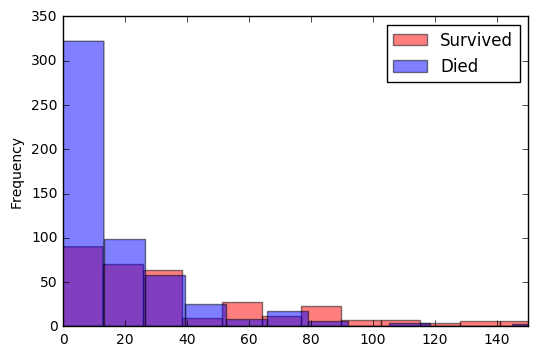
You can download the CSV from the previous step [here](https://s3.amazonaws.com/dq-content/186/submission_1.csv). When you submit it to Kaggle, you'll see that the score is 77.0%, which at the time of writing equates to jumping about 1,500 places up the leaderboard (this will vary as the leaderboard is always changing). It's only a small improvement, but we're moving in the right direction.

A lot of the gains in accuracy in machine learning come from **Feature Engineering**. Feature engineering is the practice of creating new features from your existing data.

One common way to engineer a feature is using a technique called **binning**. Binning is when you take a continuous feature, like the fare a passenger paid for their ticket, and separate it out into several ranges (or 'bins'), turning it into a categorical variable.

This can be useful when there are patterns in the data that are non-linear and you're using a linear model (like logistic regression). We actually used binning in the previous mission when we dealt with the Age column, although we didn't use the term.

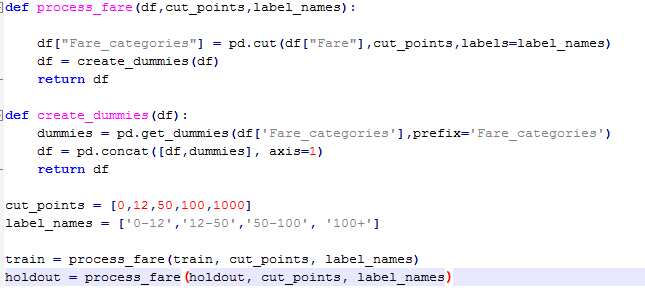
Let's look at histograms of the Fare column for passengers who died and survived, and see if there are patterns that we can use when creating our bins.



Looking at the values, it looks like we can separate the feature into four bins to capture some patterns from the data:

* 0-12
* 12-50
* 50-100
* 100+

Like in the previous mission, we can use the [pandas.cut() function](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.cut.html" \t "_blank)to create our bins.



Another way to engineer features is by extracting data from text columns. Earlier, we decided that the Name and Cabin columns weren't useful by themselves, but what if there is some data there we could extract? Let's take a look at a random sample of rows from those two columns:

|  | **Name** | **Cabin** |
| --- | --- | --- |
| **772** | Mack, Mrs. (Mary) | E77 |
| **148** | Navratil, Mr. Michel ("Louis M Hoffman") | F2 |
| **707** | Calderhead, Mr. Edward Pennington | E24 |
| **879** | Potter, Mrs. Thomas Jr (Lily Alexenia Wilson) | C50 |
| **21** | Beesley, Mr. Lawrence | D56 |
| **456** | Millet, Mr. Francis Davis | E38 |
| **97** | Greenfield, Mr. William Bertram | D10 D12 |
| **263** | Harrison, Mr. William | B94 |
| **393** | Newell, Miss. Marjorie | D36 |
| **759** | Rothes, the Countess. of (Lucy Noel Martha Dye... | B77 |

While in isolation the cabin number of each passenger will be reasonably unique to each, we can see that the format of the cabin numbers is one letter followed by two numbers. It seems like the letter is representative of the type of cabin, which could be useful data for us. We can use the pandas [Series.str accessor](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.str.html" \t "_blank) and then subset the first character using brackets:



>>> train.head()["Cabin"]

​

   0     NaN

   1     C85

   2     NaN

   3    C123

   4     NaN

   Name: Cabin, dtype: object

​

>>> train.head()["Cabin"].str[0]

​

   0    NaN

   1      C

   2    NaN

   3      C

   4    NaN

   Name: Cabin, dtype: object

Looking at the Name column, There is a title like 'Mr' or 'Mrs' within each, as well as some less common titles, like the 'Countess' from the final row of our table above. By spending some time researching the different titles, we can categorize these into six types:

* Mr
* Mrs
* Master
* Miss
* Officer
* Royalty

We can use the [Series.str.extract method](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.str.extract.html" \t "_blank) and a [regular expression](https://en.wikipedia.org/wiki/Regular_expression)to extract the title from each name and then use the [Series.map() method](http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.map.html" \t "_blank) and a predefined dictionary to simplify the titles.



titles = {

   "Mme":         "Mrs",

   "Ms":          "Mrs",

   "Mrs" :        "Mrs",

   "Countess":    "Royalty",

   "Lady" :       "Royalty"

}

​

extracted\_titles = train["Name"].str.extract(' ([A-Za-z]+)\.', expand=False)

train["Title"] = extracted\_titles.map(titles)

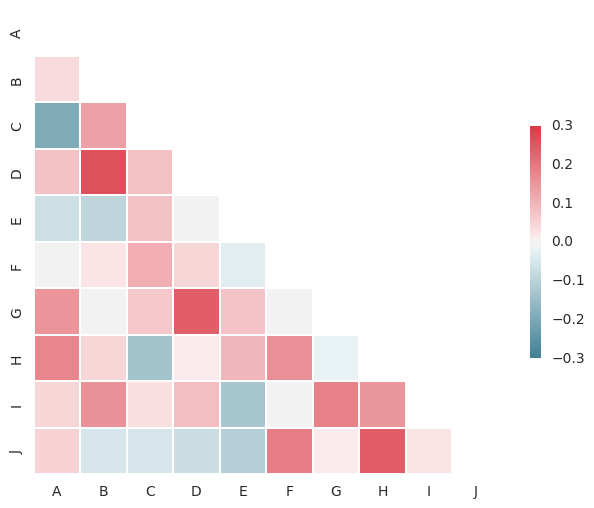
We now have 34 possible feature columns we can use to train our model. One thing to be aware of as you start to add more features is a concept called **collinearity**. Collinearity occurs where more than one feature contains data that are similar.

The effect of collinearity is that your model will overfit - you may get great results on your test data set, but then the model performs worse on unseen data (like the holdout set).

One easy way to understand collinearity is with a simple binary variable like the Sex column in our dataset. Every passenger in our data is categorized as either male or female, so 'not male' is exactly the same as 'female'.

As a result, when we created our two dummy columns from the categorical Sex column, we've actually created two columns with identical data in them. This will happen whenever we create dummy columns, and is called the [dummy variable trap](http://www.algosome.com/articles/dummy-variable-trap-regression.html). The easy solution is to choose one column to drop any time you make dummy columns.

Collinearity can happen in other places, too. A common way to spot collinearity is to plot correlations between each pair of variables in a heatmap. An example of this style of plot is below:



The darker squares, whether the darker red or darker blue, indicate pairs of columns that have higher correlation and may lead to collinearity. The easiest way to produce this plot is using the [DataFrame.corr() method](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.corr.html" \t "_blank) to produce a correlation matrix, and then use the Seaborn library's [seaborn.heatmap() function](https://seaborn.pydata.org/generated/seaborn.heatmap.html" \t "_blank) to plot the values:



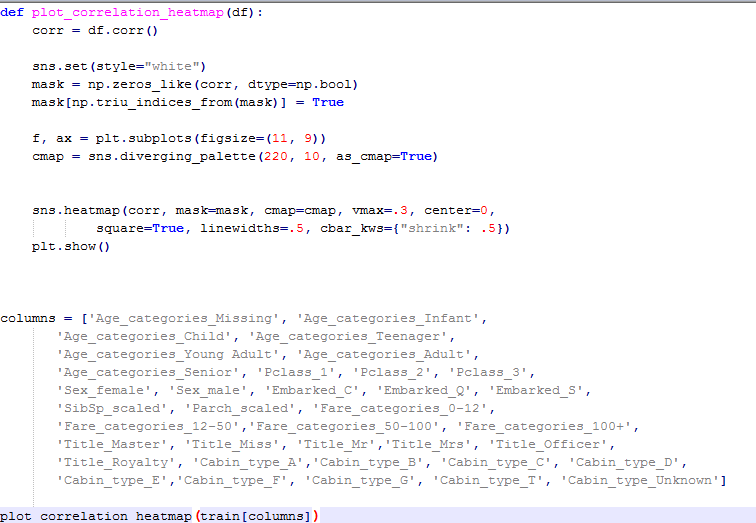
import seaborn as sns

correlations = train.corr()

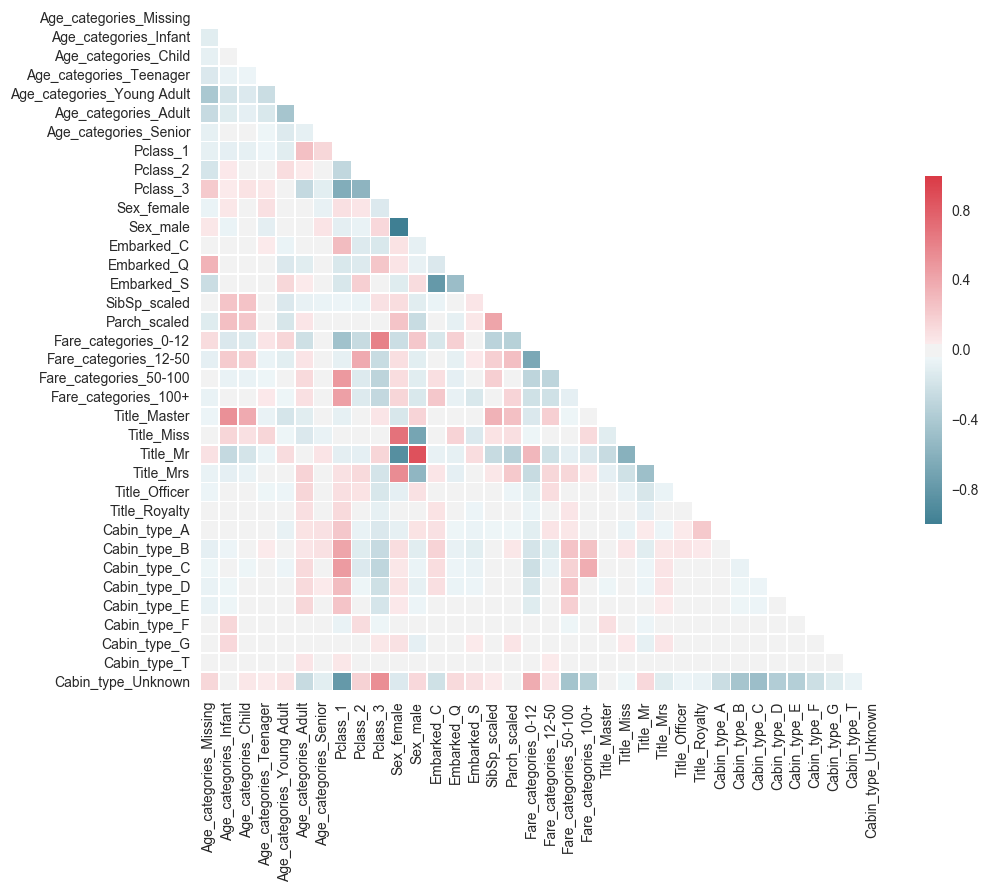
sns.heatmap(correlations)

plt.show()

The example plot above was produced using a [code example from seaborn's documentation](http://seaborn.pydata.org/examples/many_pairwise_correlations.html) which produces an correlation heatmap that is easier to interpret than the default output of heatmap(). We've created a function containing that code to make it easier for you to plot the correlations between the features in our data.



The plot we created in the previous screen is reproduced below:



We can see that there is a high correlation between Sex\_female/Sex\_male and Title\_Miss/Title\_Mr/Title\_Mrs. We will remove the columns Sex\_female and Sex\_male since the title data may be more nuanced.

Apart from that, we should remove one of each of our dummy variables to reduce the collinearity in each. We'll remove:

* Pclass\_2
* Age\_categories\_Teenager
* Fare\_categories\_12-50
* Title\_Master
* Cabin\_type\_A

In an earlier step, we manually used the logit coefficients to select the most relevant features. An alternate method is to use one of scikit-learn's inbuilt feature selection classes. We will be using the [feature\_selection.RFECV class](http://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html" \t "_blank) which performs **recursive feature elimination** with cross-validation.

The RFECV class starts by training a model using all of your features and scores it using cross validation. It then uses the logit coefficients to eliminate the least important feature, and trains and scores a new model. At the end, the class looks at all the scores, and selects the set of features which scored highest.

Like the LogisticRegression class, RFECV must first be instantiated and then fit. The first parameter when creating the RFECV object must be an estimator, and we need to use the cv parameter to specific the number of folds for cross-validation.



from sklearn.feature\_selection import RFECV

lr = LogisticRegression()

selector = RFECV(lr,cv=10)

selector.fit(all\_X,all\_y)

Once the RFECV object has been fit, we can use the RFECV.support\_attribute to access a boolean mask of True and False values which we can use to generate a list of optimized columns:



optimized\_columns = all\_X.columns[selector.support\_]

The RFECV() selector returned only four columns:



['SibSp\_scaled', 'Title\_Mr', 'Title\_Officer', 'Cabin\_type\_Unknown']

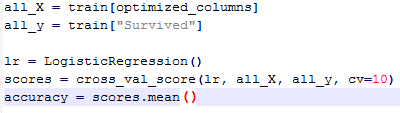
Let's train a model using cross validation using these columns and check the score.

The RFECV() selector returned only four columns:



['SibSp\_scaled', 'Title\_Mr', 'Title\_Officer', 'Cabin\_type\_Unknown']

Let's train a model using cross validation using these columns and check the score.



This four-feature model scores 82.3%, a modest improvement compared to the 81.5% from our earlier model. Let's train these columns on the holdout set, save a submission file and see what score we get from Kaggle.

To improve this score, we can use Model Selection. The process of selecting the algorithm which gives the best predictions for your data is called **model selection**.

In this mission, we're going work with two new algorithms: k-nearest neighbors and random forests.

"fit" fits a model against some training data so you can later do a predict with some different data

Cross\_val\_score just gives you a quick measure of the accuracy of the model for Model Selection. Use **cross validation** to train and test our model on different splits of our data, and then average the accuracy scores.

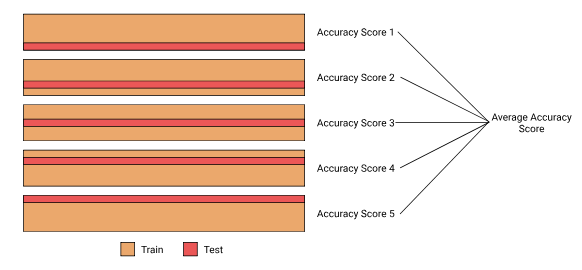
\_coef is the gradient, m, y = mx + c. It’s the change in train\_y per unit change in train\_X. e.g taking sex\_male. If I change it from 0 to 1, what the rate of change in ‘survival’. The the value of ‘survival’ change (i.e. the greater the gradient), then the more of an effect the feature sex\_male have on ‘survival’, and hence is significant feature to have or keep. If instead, changing ‘sex\_male’ has little or no change in ‘survival’ (i.e. lower gradient), then we can use Feature Selection and remove that feature from or model

Feature Selection (using coef\_ to remove feature) and Feature Engineering (using binning and RFECV)  to make our model more accurate and then we test the accuracy of the model using Cross Validation (cross\_val\_score)

**Cross Validation**

80 - 20 split in training and test data can cause overfitting. Given that this data set is quite small, there is a good chance that our model is overfitting, and will not perform as well on totally unseen data.

To give us a better understanding of the real performance of our model, we can use a technique called **cross validation** to train and test our model on different splits of our data, and then average the accuracy scores.



The most common form of cross validation, and the one we will be using, is called **k-fold** cross validation. 'Fold' refers to each different iteration that we train our model on, and 'k' just refers to the number of folds. In the diagram above, we have illustrated k-fold validation where k is 5.

We will use scikit-learn's [model\_selection.cross\_val\_score()function](http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html" \l "sklearn.model_selection.cross_val_score" \t "_blank) to automate the proces