LDSSA

SLU16 - Data Sufficiency and Selection

July 6th and 7th, 2019

1. Introduction

Motivation

So why should we care about data sufficiency and selection?

- It will help us control complexity in our models which controls for unexpected behavior and interpretability
- It will help us avoid overfitting our models (BIG DEAL)
- It will help us understand how the nature of a trained model changes when you add data or features

Overview

- Univariate Selection
- Feature Importances
- Correlation
- Learning Curves

2. Topic Explanation

How will we do this?

We are going to take a little journey through working on the kaggle titanic competition

We're going to select different subsets of the training data, train models on them, and then inspect these models to see how they behave

Spoiler: the leaderboard standing is going to matter a lot less than you think and that's for a reason

Please, if you don’t mind

Let’s suspend disbelief for a bit here - some of the problems I will present have very easy ways out of them but I’m not going to take them for argument sake

This is because the concepts will still hold in different situations where there may not be an easy way out

Imagine

Okay say that your big bossman gives you a dataset and tells you he needs a predictive model.

"Yes sir!" you say, "I'll throw all of the features into model, call fit() and then we'll be good to deploy!"

Is this a good idea?

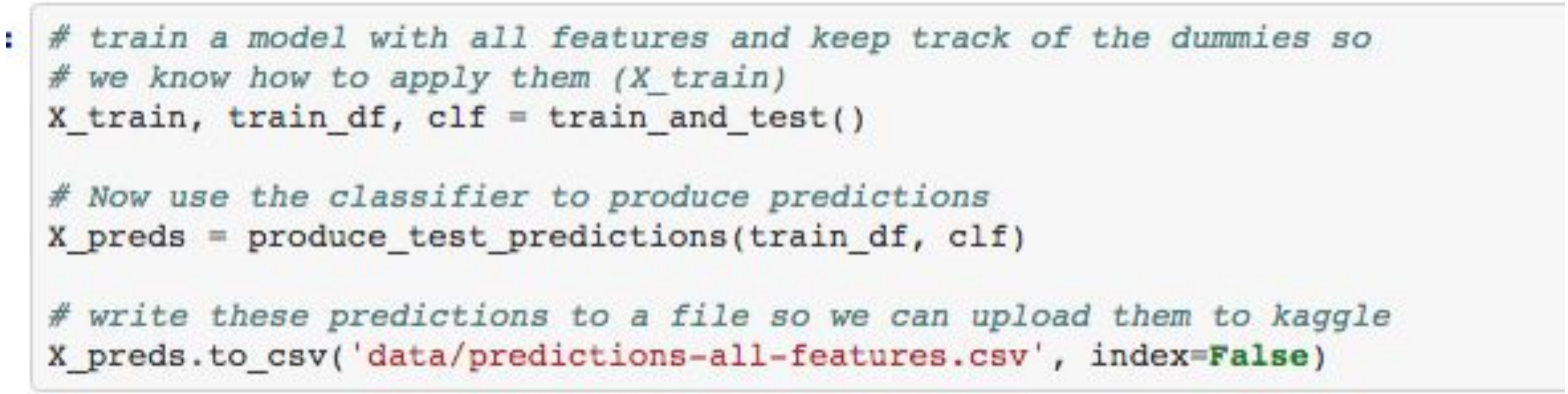
No, it is not a good idea. You have no idea what your model learned and how over (or under) fit it might be. Luckily we have a set of techniques in data sufficency and selection to help you:

- control complexity in models which controls for unexpected behavior and interpretability
- Determine if you should use a feature
- Determine if you have too many features.
- Explain if your model / is it behaving as expected.
- Check if your model is over fit (can't repeat this enough time)

Why you ask?

Let’s get our hands dirty and find out

We're going to use 2 functions heavily in a flow that looks like this

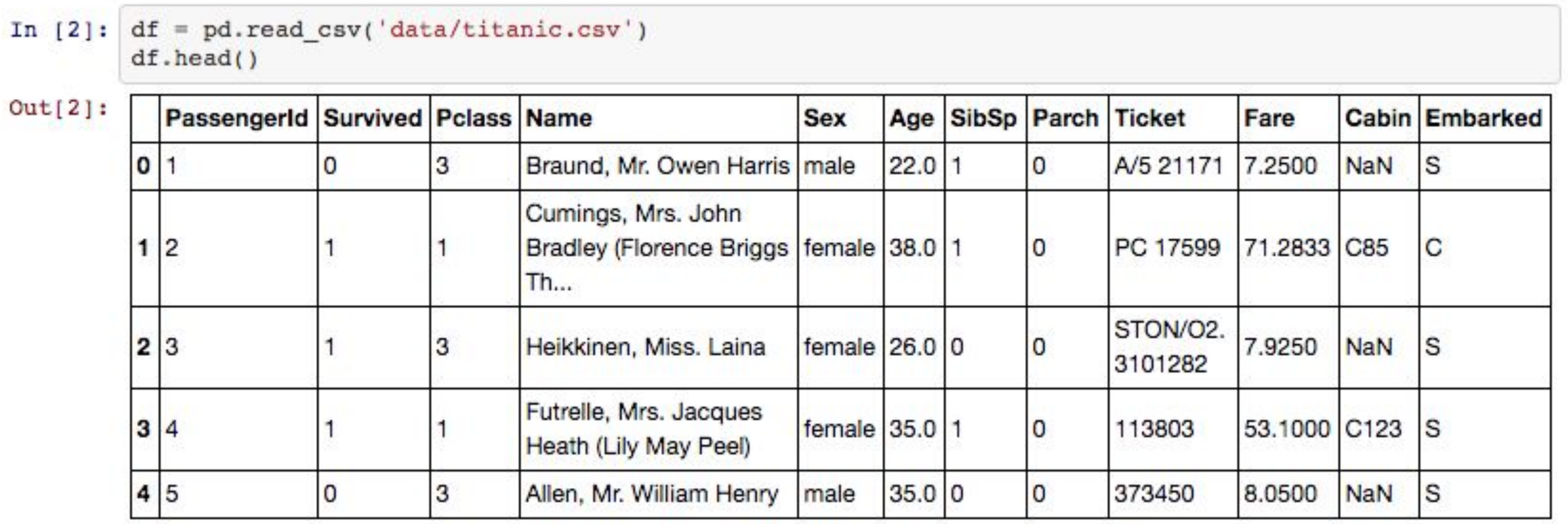


```
# train a model with all features and keep track of the dummies so
# we know how to apply them (X_train)
X_train, train_df, clf = train_and_test()

# Now use the classifier to produce predictions
X_preds = produce_test_predictions(train_df, clf)

# write these predictions to a file so we can upload them to kaggle
X_preds.to_csv('data/predictions-all-features.csv', index=False)
```

Don't worry too much about the contents of them, just worry about the concepts

Let’s get started

You know the drill *yawn*

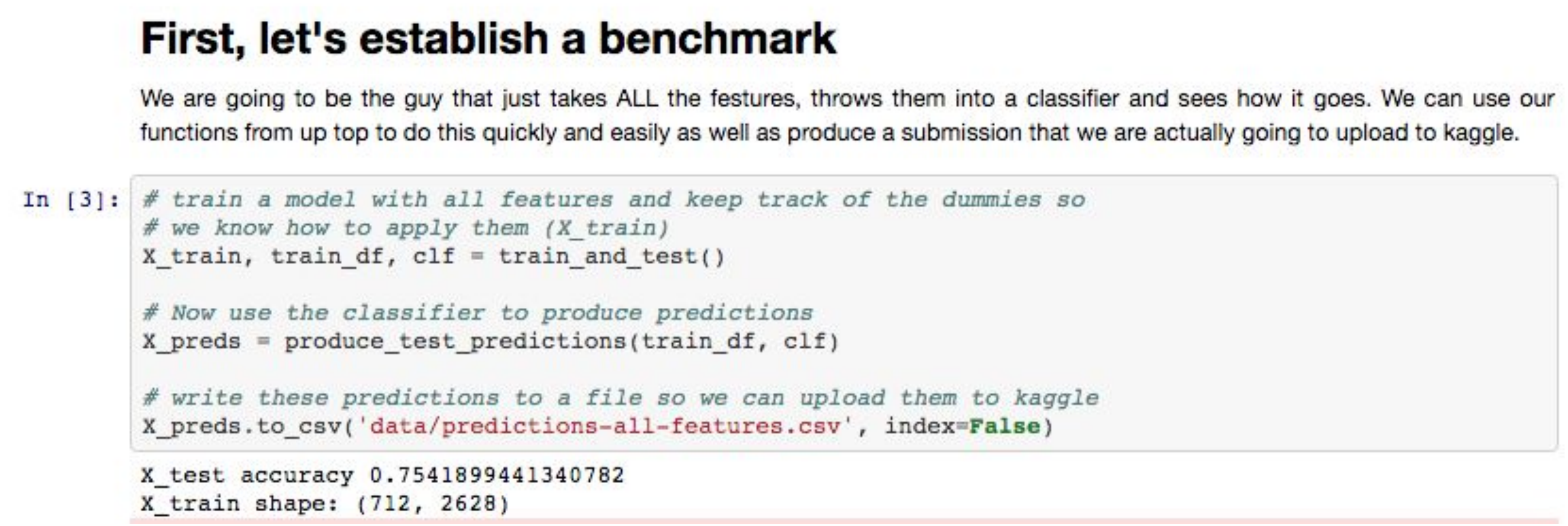


```
In [2]: df = pd.read_csv('data/titanic.csv')
        df.head()
```

Out[2]:

	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

First try



First, let's establish a benchmark

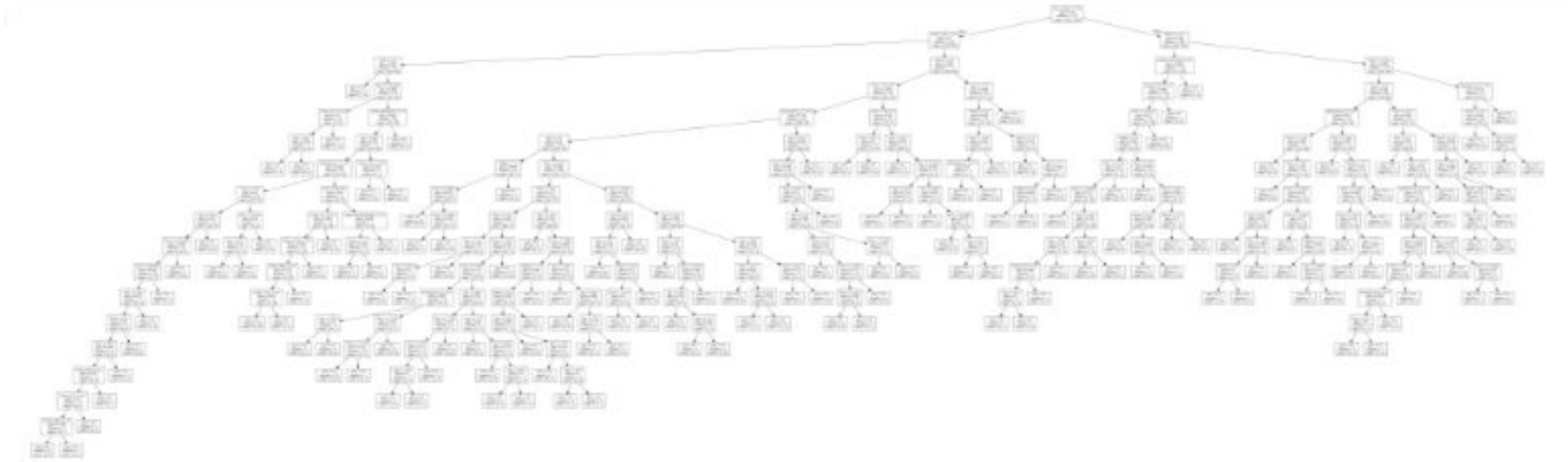
We are going to be the guy that just takes ALL the festures, throws them into a classifier and sees how it goes. We can use our functions from up top to do this quickly and easily as well as produce a submission that we are actually going to upload to kaggle.

In [3]:

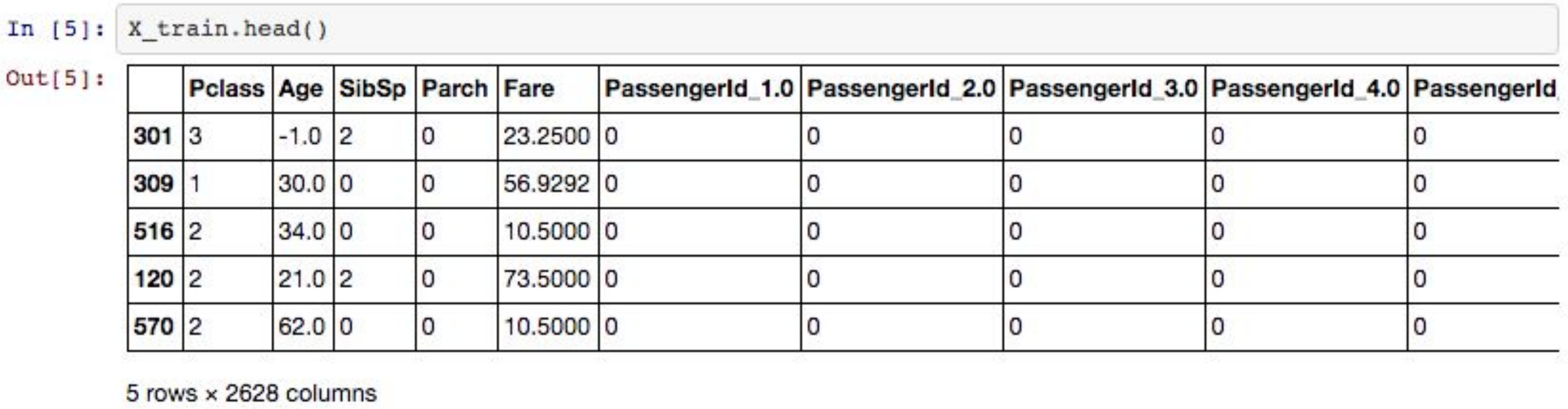
```
# train a model with all features and keep track of the dummies so
# we know how to apply them (X_train)
X_train, train_df, clf = train_and_test()

# Now use the classifier to produce predictions
X_preds = produce_test_predictions(train_df, clf)

# write these predictions to a file so we can upload them to kaggle
X_preds.to_csv('data/predictions-all-features.csv', index=False)
```

```
X_test accuracy 0.7541899441340782
X_train shape: (712, 2628)
```

Here’s the tree



Dimensionality



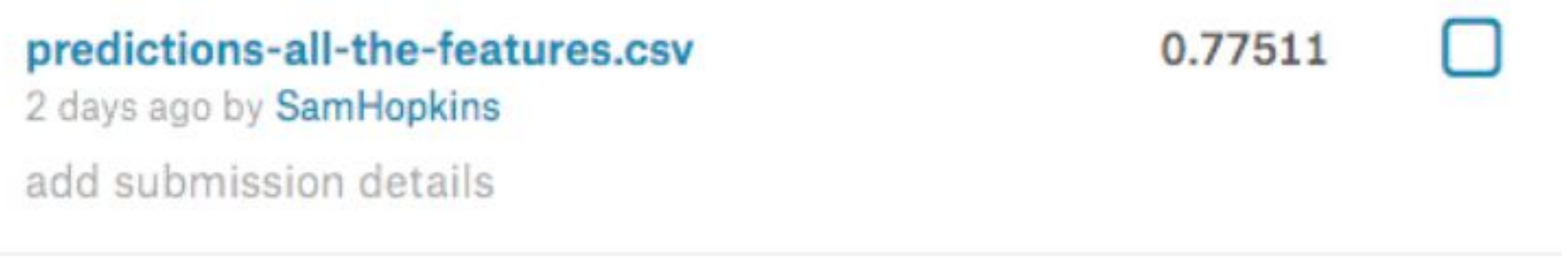
```
In [5]: X_train.head()
```

Out[5]:

	Pclass	Age	SibSp	Parch	Fare	PassengerId_1.0	PassengerId_2.0	PassengerId_3.0	PassengerId_4.0	PassengerId
301	3	-1.0	2	0	23.2500	0	0	0	0	0
309	1	30.0	0	0	56.9292	0	0	0	0	0
516	2	34.0	0	0	10.5000	0	0	0	0	0
120	2	21.0	2	0	73.5000	0	0	0	0	0
570	2	62.0	0	0	10.5000	0	0	0	0	0

5 rows × 2628 columns

The results

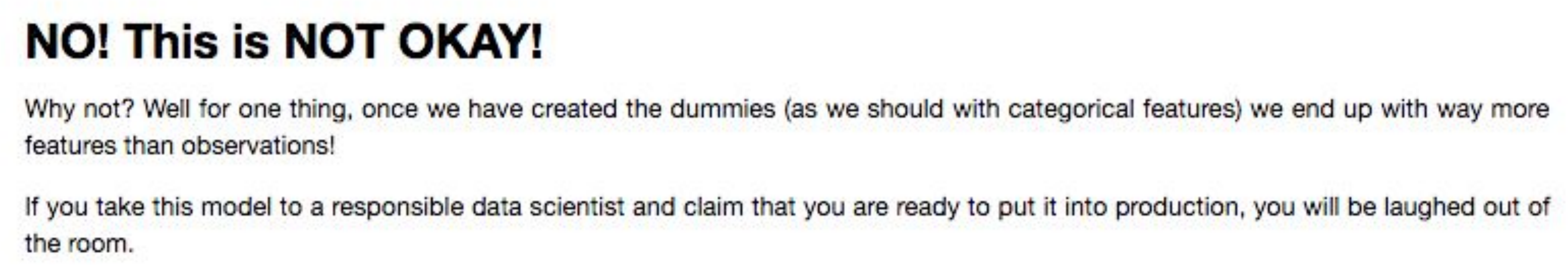


predictions-all-the-features.csv
2 days ago by SamHopkins
add submission details

0.77511

Is this okay?

No! No! No! No!

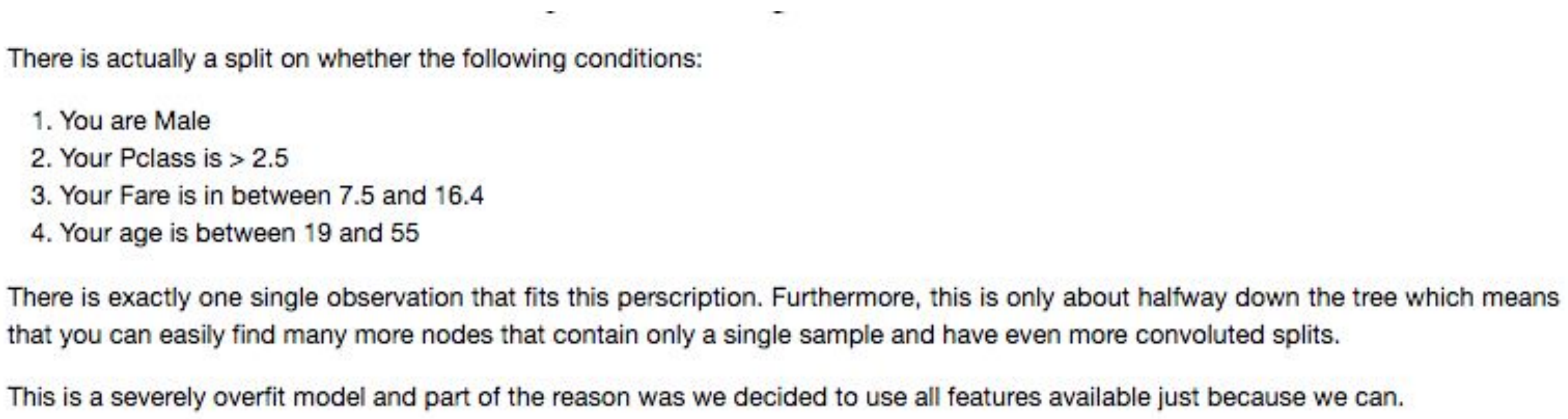


NO! This is NOT OKAY!

Why not? Well for one thing, once we have created the dummies (as we should with categorical features) we end up with way more features than observations!

If you take this model to a responsible data scientist and claim that you are ready to put it into production, you will be laughed out of the room.

Let's look at the model



There is actually a split on whether the following conditions:

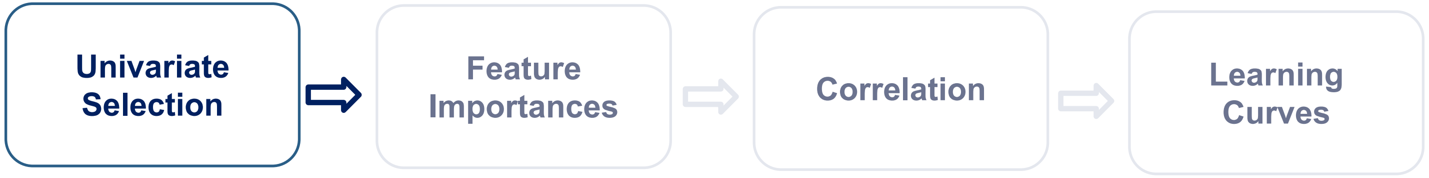
1. You are Male
2. Your Pclass is > 2.5
3. Your Fare is in between 7.5 and 16.4
4. Your age is between 19 and 55

There is exactly one single observation that fits this perscription. Furthermore, this is only about halfway down the tree which means that you can easily find many more nodes that contain only a single sample and have even more convoluted splits.

This is a severely overfit model and part of the reason was we decided to use all features available just because we can.

So what should we do?

Use our brains and select some features to get rid of!

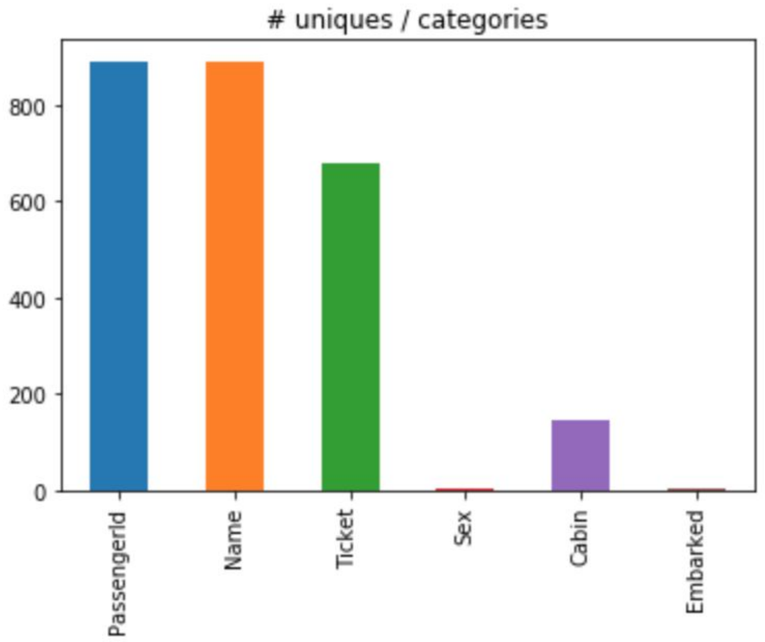

Univariate Selection
Feature Importances
Correlation
Learning Curves

Okay, so let's actually select some features

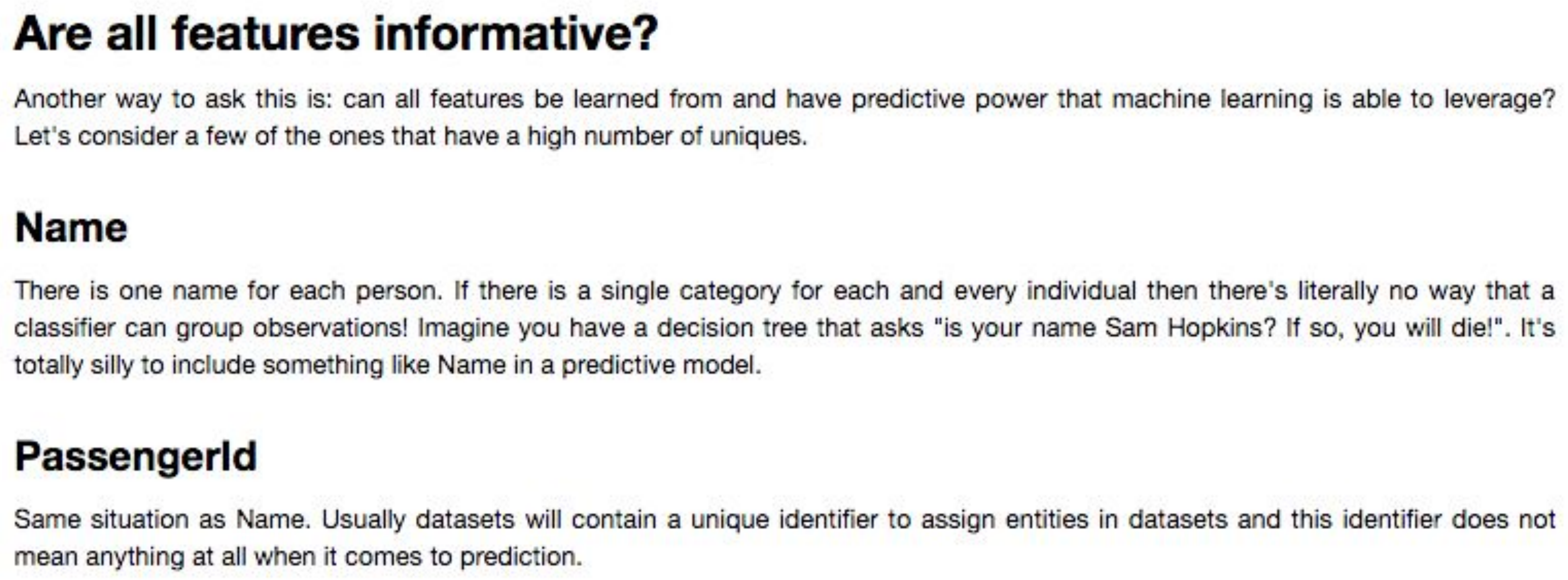
The first thing that we can do is to find the low-hanging fruit with a little bit of Single Factor Analysis.

A couple of good heuristics is to look for categorical features that contain all unique values. Let's create a new series whose only job is to count the number of unique values (a.k.a categories) for each categorical value:

```
df.loc[:, categoricals].nunique().plot(kind='bar', title='# uniques / categories');
```



Which to remove?



Are all features informative?

Another way to ask this is: can all features be learned from and have predictive power that machine learning is able to leverage? Let's consider a few of the ones that have a high number of uniques.

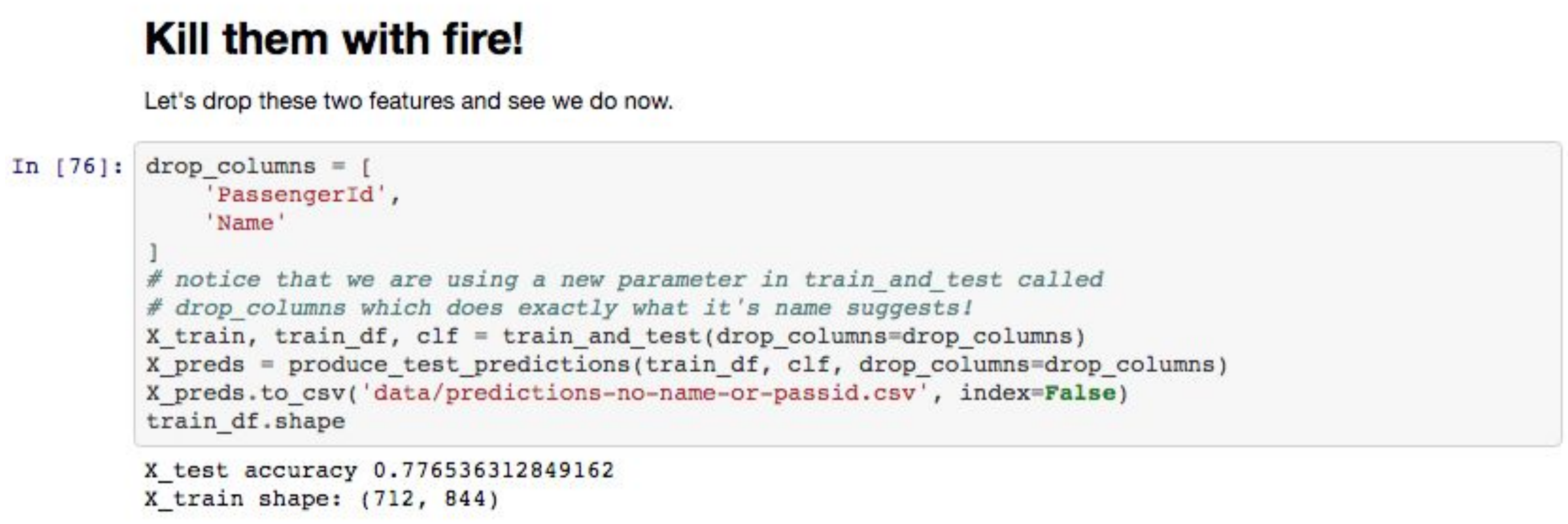
Name

There is one name for each person. If there is a single category for each and every individual then there's literally no way that a classifier can group observations! Imagine you have a decision tree that asks "is your name Sam Hopkins? If so, you will die!". It's totally silly to include something like Name in a predictive model.

PassengerId

Same situation as Name. Usually datasets will contain a unique identifier to assign entities in datasets and this identifier does not mean anything at all when it comes to prediction.

Same routine as before but without some features



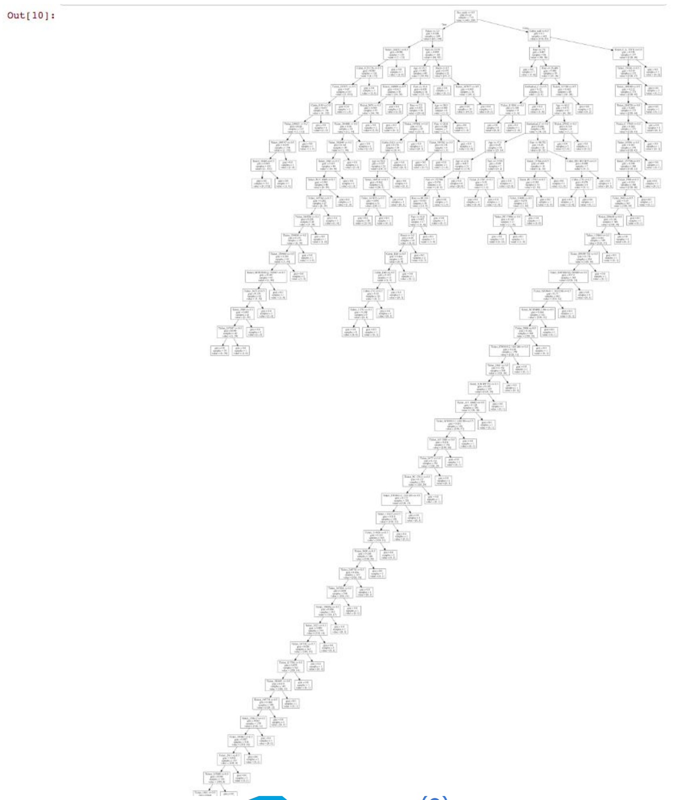
Kill them with fire!

Let's drop these two features and see we do now.

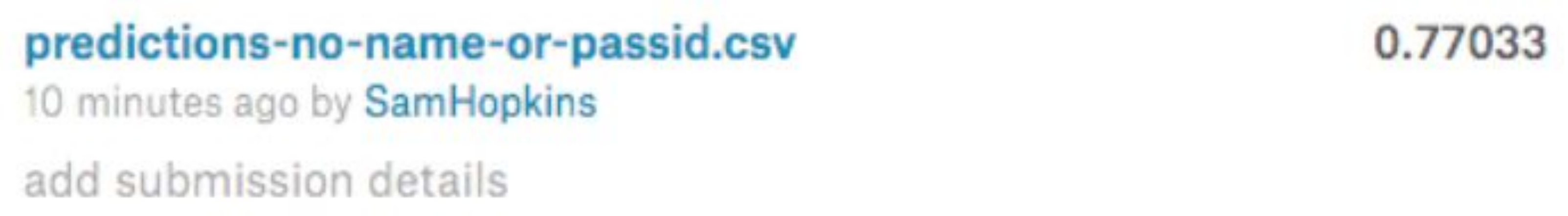
```
In [76]: drop_columns = [
             'PassengerId',
             'Name'
         ]
         # notice that we are using a new parameter in train_and_test called
         # drop_columns which does exactly what it's name suggests!
         X_train, train_df, clf = train_and_test(drop_columns=drop_columns)
         X_preds = produce_test_predictions(train_df, clf, drop_columns=drop_columns)
         X_preds.to_csv('data/predictions-no-name-or-passid.csv', index=False)
         train_df.shape
```

```
X_test accuracy 0.776536312849162
X_train shape: (712, 844)
```

And we got ourselves a funky new tree



And the new results


predictions-no-name-or-passid.csv
10 minutes ago by SamHopkins
add submission details
0.77033

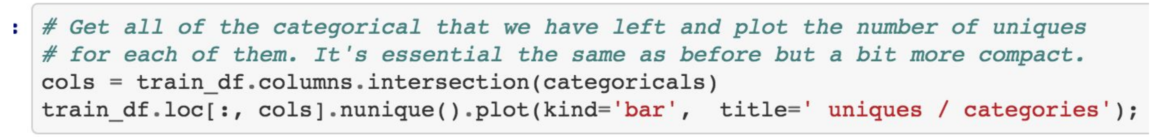
Old score was 0.77511

This is basically the same score - 0.005 accuracy only matters in kaggle, not the real world

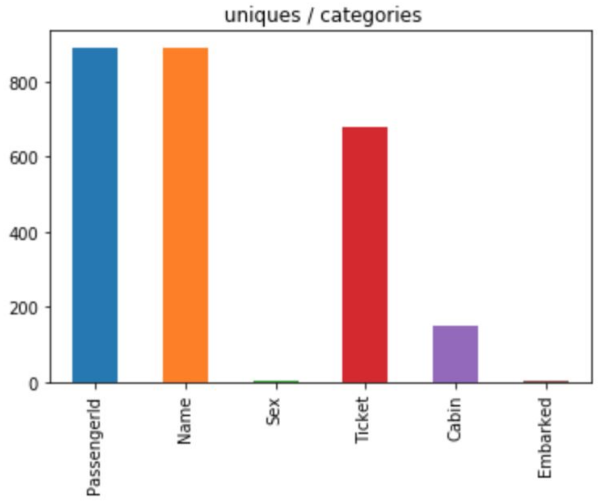
We’re moving in the right direction

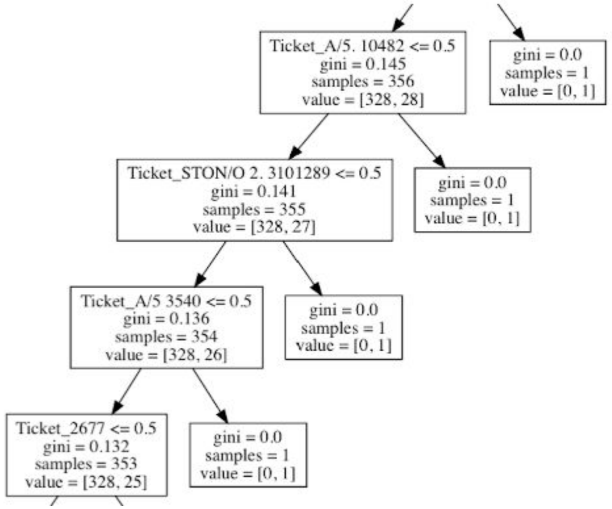
- Although simpler, the tree is still pretty insane
- We didn’t sacrifice any performance
- This is a huge deal! Reducing complexity without sacrificing performance is a HUGE WIN in the real world when you have stakeholders such as auditors involved.

Let's keep on the overfiting problem



```
# Get all of the categorical that we have left and plot the number of uniques
# for each of them. It's essential the same as before but a bit more compact.
cols = train_df.columns.intersection(categoricals)
train_df.loc[:, cols].nunique().plot(kind='bar',  title=' uniques / categories');
```





Should we kill the Ticket feature?

We know that we seem to be overfitting on it but it might be an important feature that we could preprocess a bit and keep around.

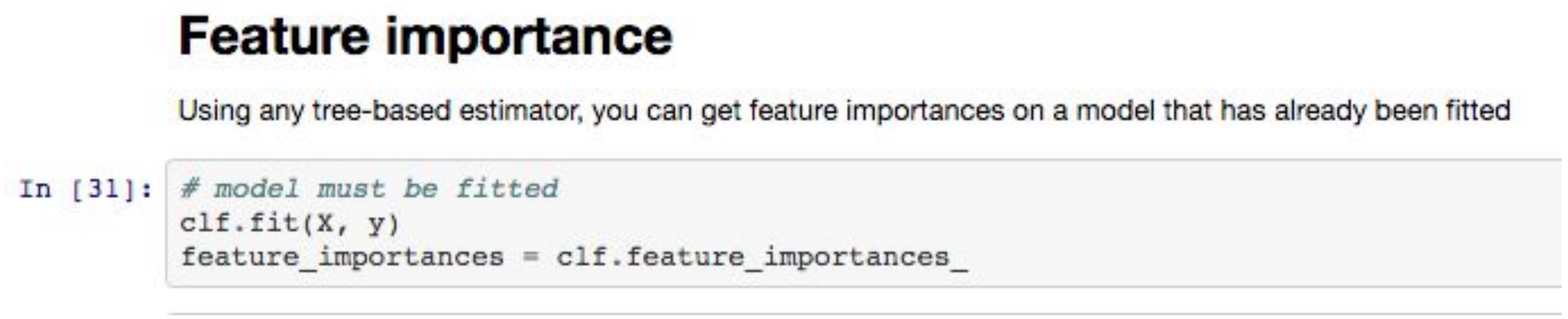
Can we get higher level view on how much of a role it is playing in our classifier before we take the decision to kill it with fire?

Univariate Selection ⇨ **Feature Importances** ⇨ Correlation ⇨ Learning Curves

We can with feature_importances_ and coef_

feature_importances is a very useful attribute that you can use to understand how much the **tree-based** and **ensemble** models rely on each feature when making predictions. There is an analogous (but not the same) feature called coef_ on **linear** models that we can also look at.

On members of the scikit.tree and some from the scikit.ensemble packages



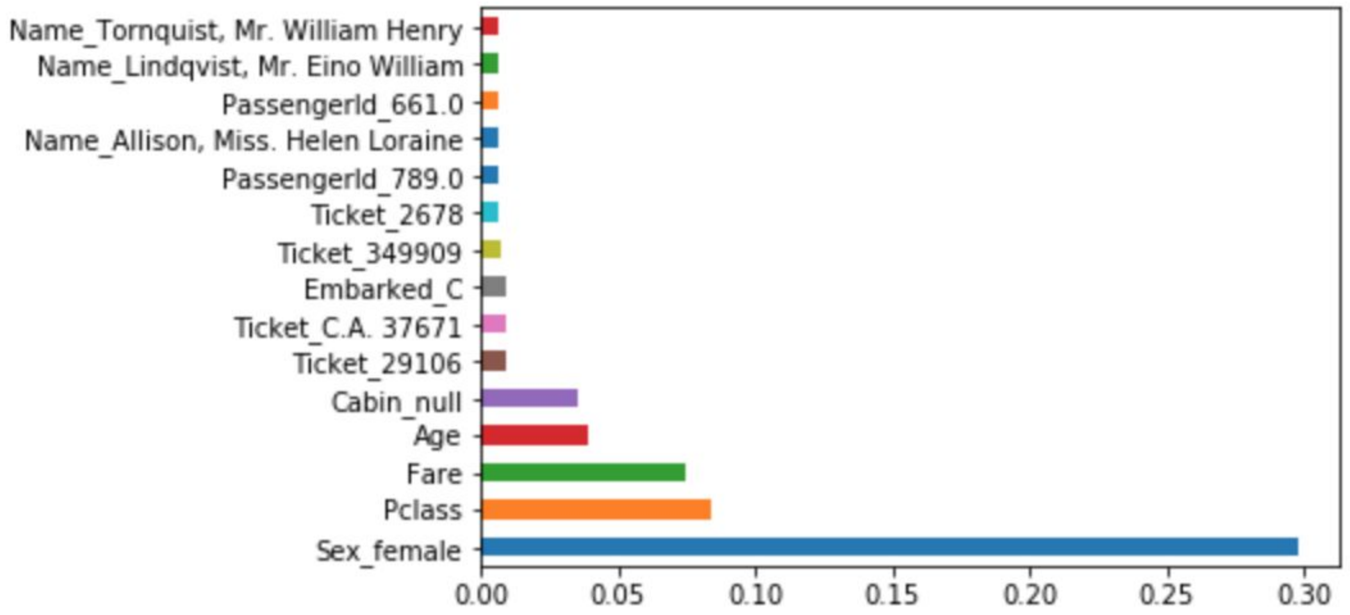
Feature importance

Using any tree-based estimator, you can get feature importances on a model that has already been fitted

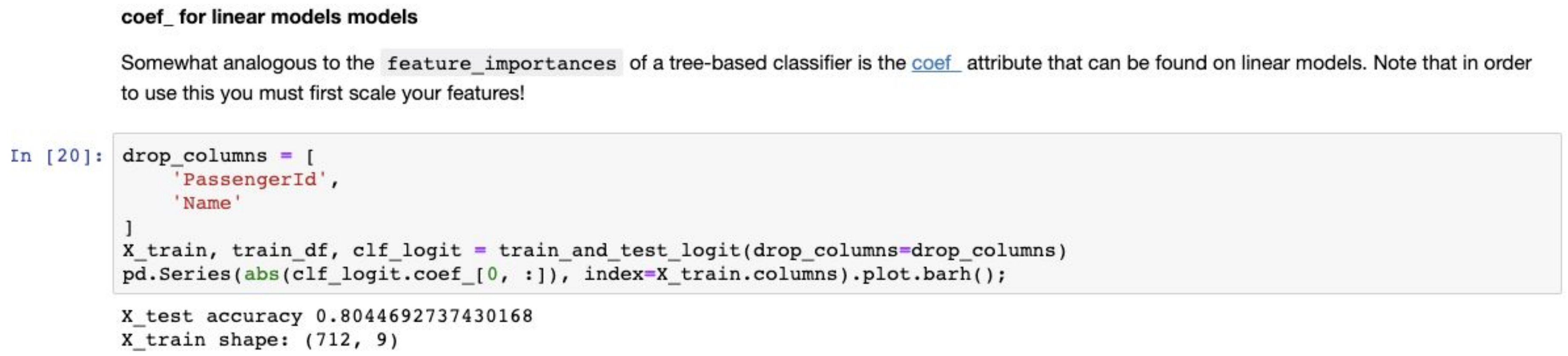
```
In [31]: # model must be fitted
         clf.fit(X, y)
         feature_importances = clf.feature_importances_
```

Plotting them can be very useful

```
pd.Series(clf.feature_importances_, index=X_train.columns).nlargest(15).plot.barh();
```



On linear models



coef_ for linear models models

Somewhat analogous to the `feature_importances` of a tree-based classifier is the coef_ attribute that can be found on linear models. Note that in order to use this you must first scale your features!

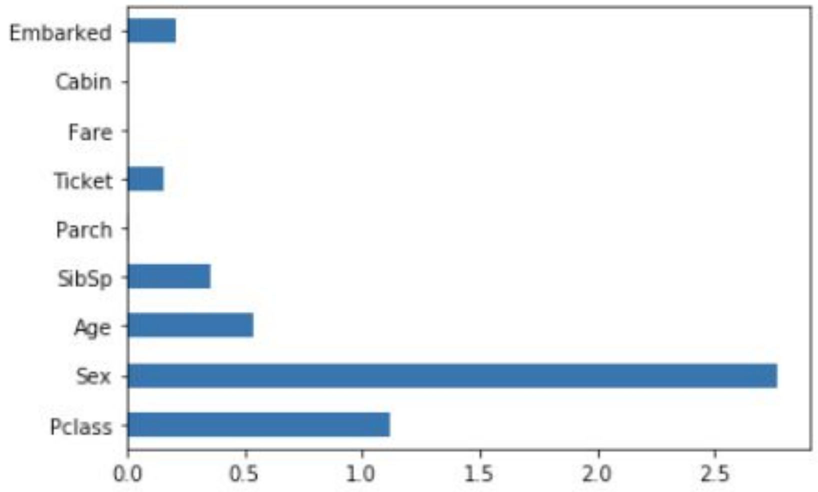
In [20]:
```
drop_columns = [
    'PassengerId',
    'Name'
]
X_train, train_df, clf_logit = train_and_test_logit(drop_columns=drop_columns)
pd.Series(abs(clf_logit.coef_[0, :]), index=X_train.columns).plot.barh();
```

```
X_test accuracy 0.8044692737430168
X_train shape: (712, 9)
```

Plotting them can be very useful

```
pd.Series(abs(clf_logit.coef_[0, :]), index=X_train.columns).plot.barh();
```

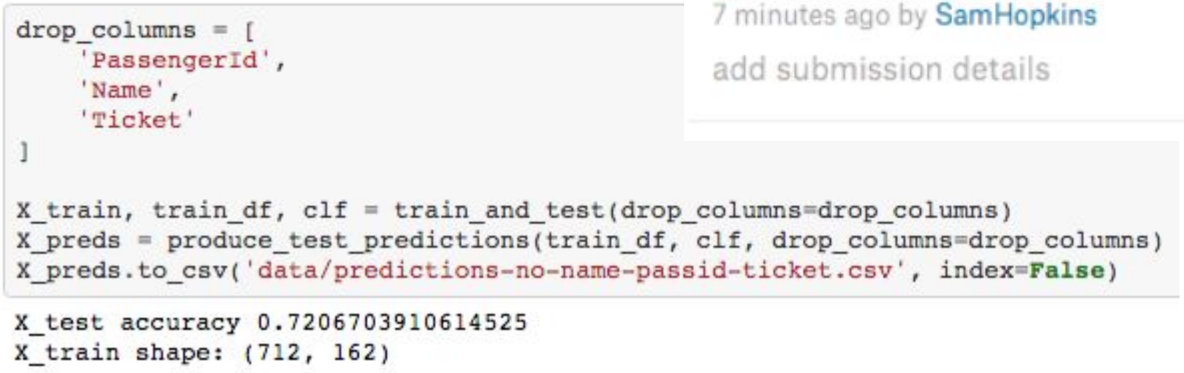
```
X_test accuracy 0.8044692737430168
X_train shape: (712, 9)
```



Kill with fire!

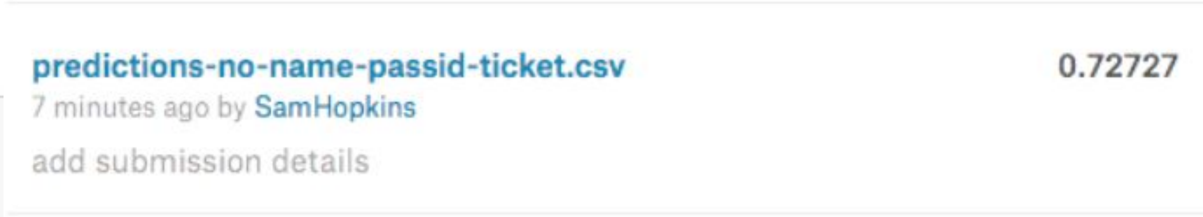
Seems like in the grand scheme of things, it could totally be worth the experiment to get rid of it in order to win a victory in the battle against complexity

How we lookin?



```
: drop_columns = [
      'PassengerId',
      'Name',
      'Ticket'
  ]

  X_train, train_df, clf = train_and_test(drop_columns=drop_columns)
  X_preds = produce_test_predictions(train_df, clf, drop_columns=drop_columns)
  X_preds.to_csv('data/predictions-no-name-passid-ticket.csv', index=False)
```

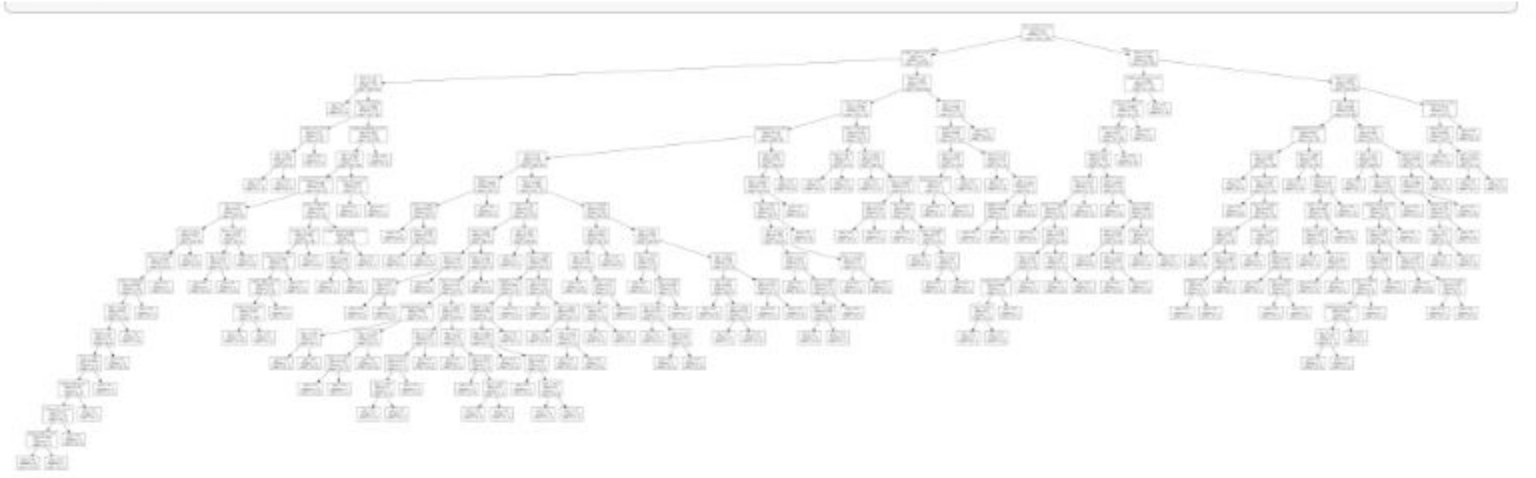
```
X_test accuracy 0.7206703910614525
X_train shape: (712, 162)
```

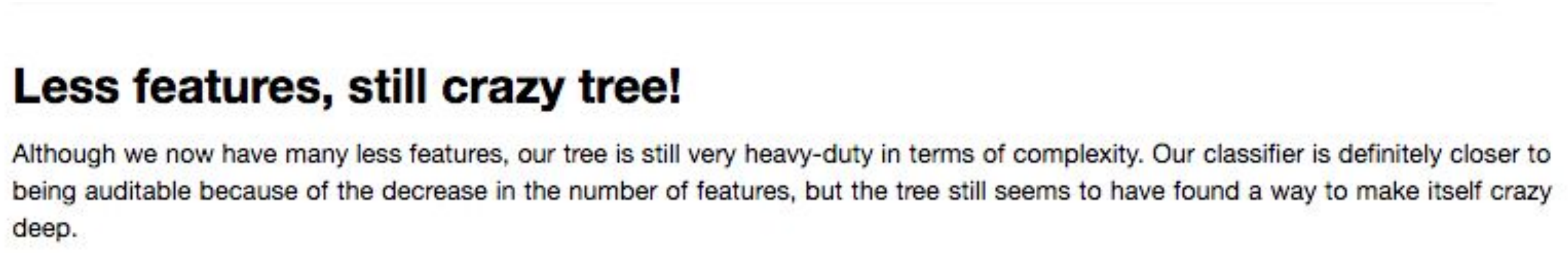


predictions-no-name-passid-ticket.csv 0.72727

7 minutes ago by SamHopkins

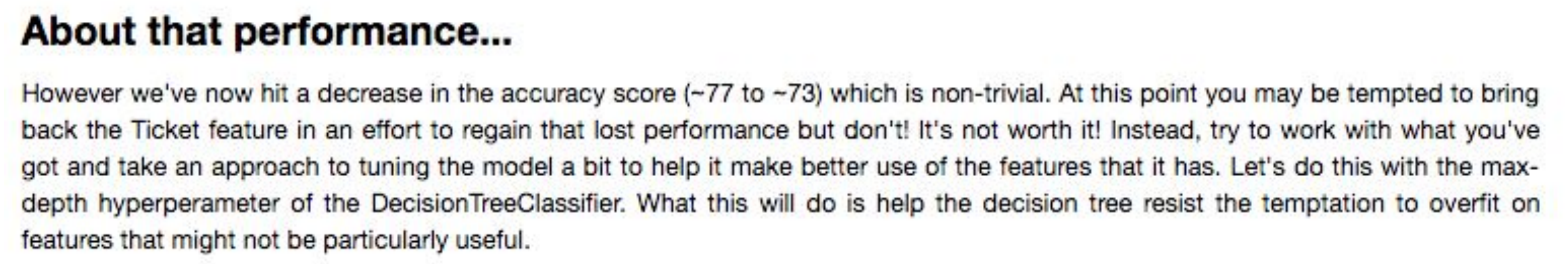
add submission details





Less features, still crazy tree!

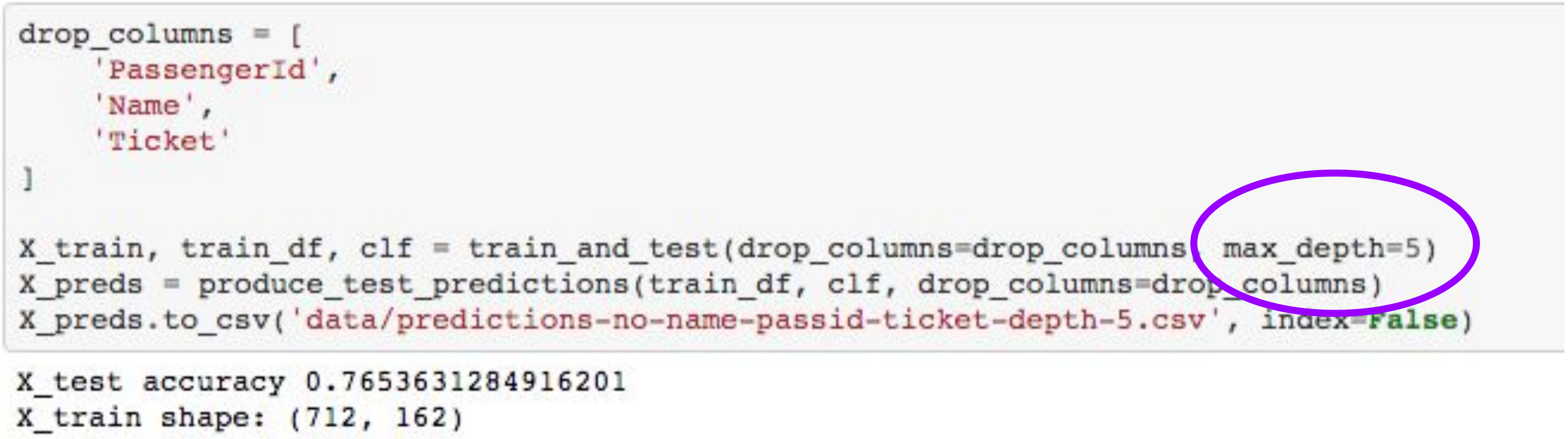
Although we now have many less features, our tree is still very heavy-duty in terms of complexity. Our classifier is definitely closer to being auditable because of the decrease in the number of features, but the tree still seems to have found a way to make itself crazy deep.



About that performance...

However we've now hit a decrease in the accuracy score (~77 to ~73) which is non-trivial. At this point you may be tempted to bring back the Ticket feature in an effort to regain that lost performance but don't! It's not worth it! Instead, try to work with what you've got and take an approach to tuning the model a bit to help it make better use of the features that it has. Let's do this with the max-depth hyperperameter of the DecisionTreeClassifier. What this will do is help the decision tree resist the temptation to overfit on features that might not be particularly useful.

Let’s do a quick tune to try and help our model reduce it’s temptation to overfit



```
drop_columns = [
    'PassengerId',
    'Name',
    'Ticket'
]

X_train, train_df, clf = train_and_test(drop_columns=drop_columns, max_depth=5)
X_preds = produce_test_predictions(train_df, clf, drop_columns=drop_columns)
X_preds.to_csv('data/predictions-no-name-passid-ticket-depth-5.csv', index=False)
```

```
X_test accuracy 0.7653631284916201
X_train shape: (712, 162)
```

Hooray! On our test set, the improvement is back up! Let's make a submission and see how that goes!

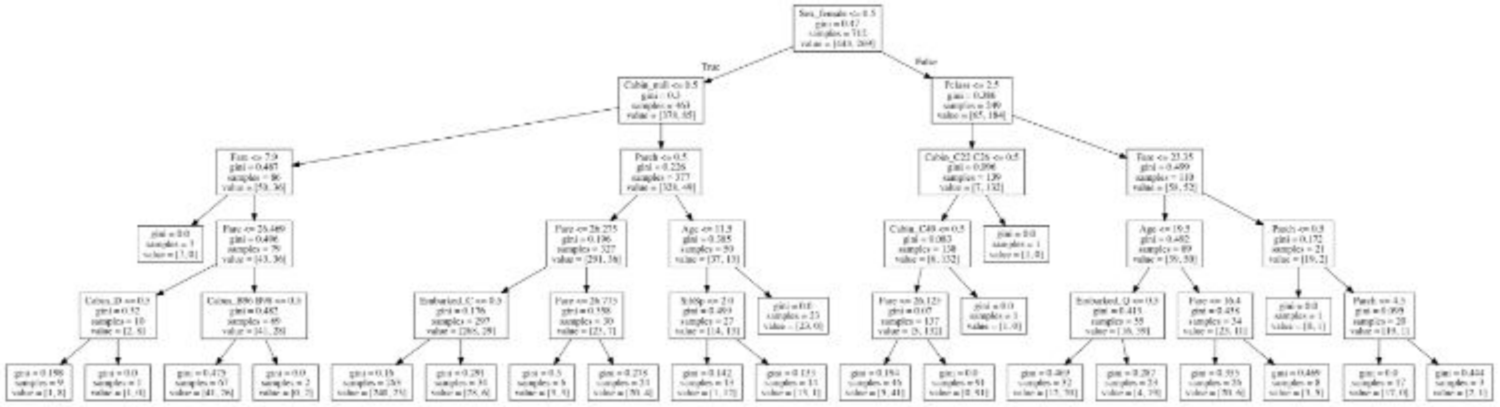
```
Image('media/kaggle-submission-no-name-passid-ticket-depth-5.png', width=800)
```

predictions-no-name-passid-ticket-depth-5.csv 0.77511

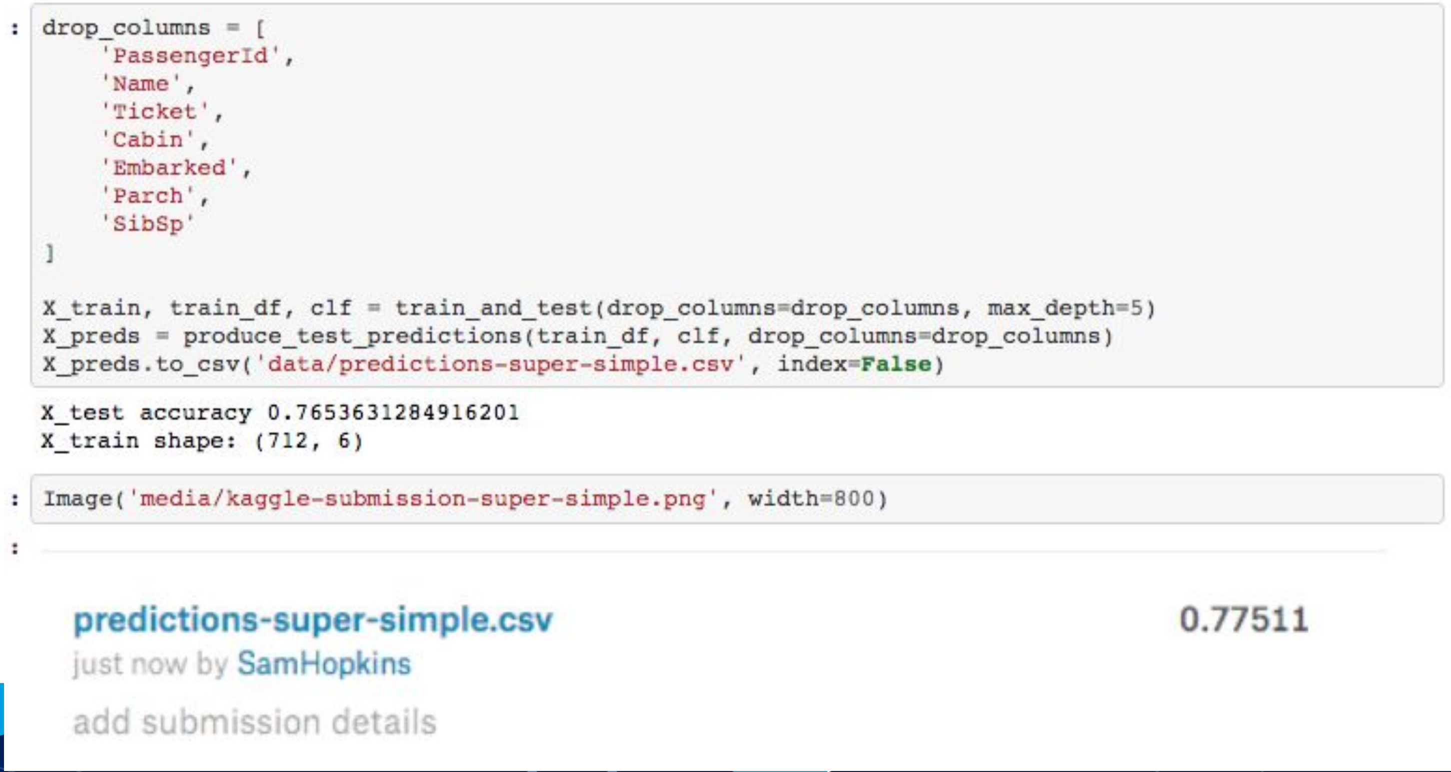
a few seconds ago by SamHopkins

add submission details

We're back in business in terms of performance, baby! And we've got a much less offensive tree! Whew, good thing we didn't succomb to the temptation of adding more features for a quick win on a kaggle metric!



Why stop now?



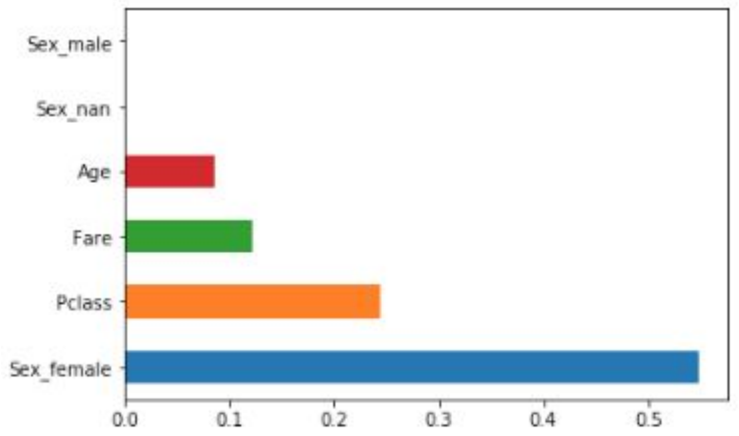
```
: drop_columns = [
      'PassengerId',
      'Name',
      'Ticket',
      'Cabin',
      'Embarked',
      'Parch',
      'SibSp'
  ]

  X_train, train_df, clf = train_and_test(drop_columns=drop_columns, max_depth=5)
  X_preds = produce_test_predictions(train_df, clf, drop_columns=drop_columns)
  X_preds.to_csv('data/predictions-super-simple.csv', index=False)

  X_test accuracy 0.7653631284916201
  X_train shape: (712, 6)

: Image('media/kaggle-submission-super-simple.png', width=800)

:
```

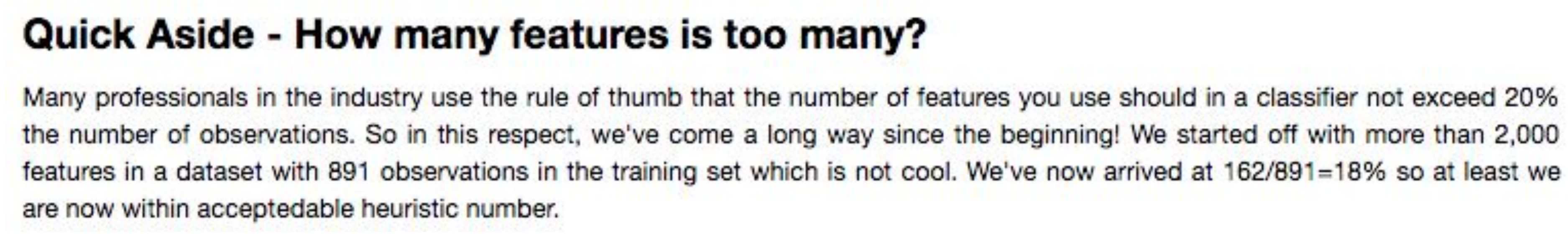
predictions-super-simple.csv 0.77511

just now by SamHopkins

add submission details

Feature importances with 6 features left





Quick Aside - How many features is too many?

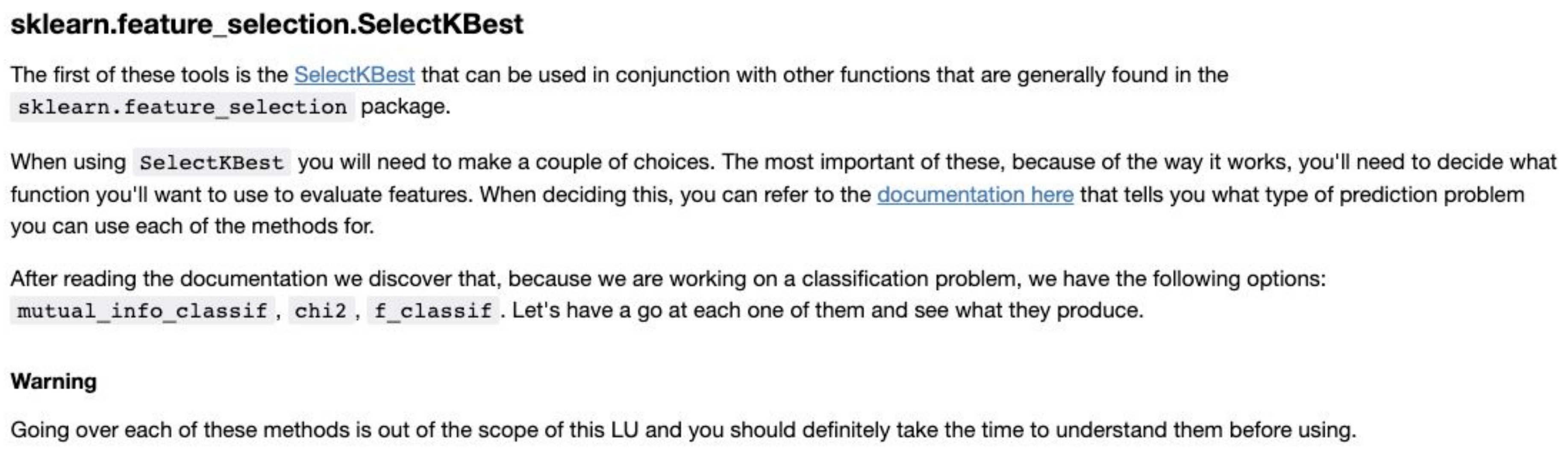
Many professionals in the industry use the rule of thumb that the number of features you use should in a classifier not exceed 20% the number of observations. So in this respect, we've come a long way since the beginning! We started off with more than 2,000 features in a dataset with 891 observations in the training set which is not cool. We've now arrived at 162/891=18% so at least we are now within acceptedable heuristic number.

$$\# \text{ features} / \# \text{ observations} < 0.2$$

Higher Level Tools

We have been using directly a few methods such as single factor analysis, feature importances, and coef_. There are a few additional tools that leverage, in a nicer api, a few tools that we are now familiar with. It’s good for keeping your code clean!

SelectKBest



sklearn.feature_selection.SelectKBest

The first of these tools is the SelectKBest that can be used in conjunction with other functions that are generally found in the `sklearn.feature_selection` package.

When using `SelectKBest` you will need to make a couple of choices. The most important of these, because of the way it works, you'll need to decide what function you'll want to use to evaluate features. When deciding this, you can refer to the documentation here that tells you what type of prediction problem you can use each of the methods for.

After reading the documentation we discover that, because we are working on a classification problem, we have the following options: `mutual_info_classif` , `chi2` , `f_classif` . Let's have a go at each one of them and see what they produce.

Warning

Going over each of these methods is out of the scope of this LU and you should definitely take the time to understand them before using.

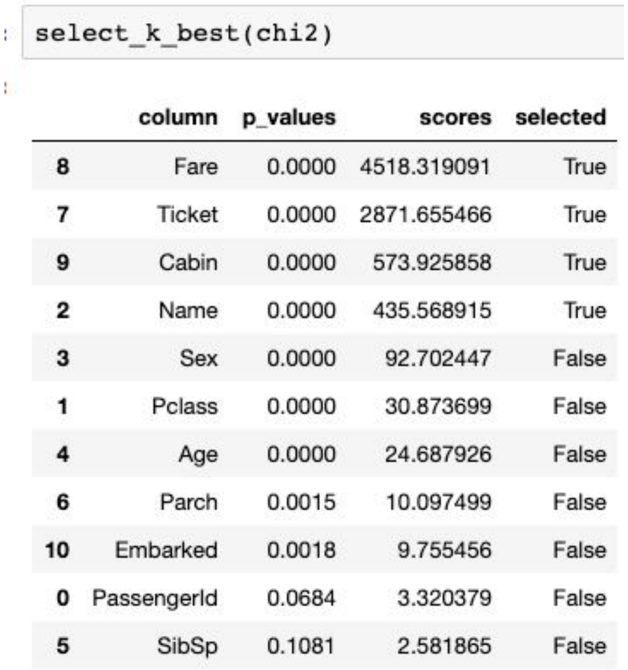
SelectKBest Usage

Well this is interesting and is a great example of what happens if you were to just blindly use a methods without taking the time to understand how they work. It's interesting to see how all of them selected `Fare` and it's equally interesting to see some of the other features that have been selected.

The statistical tests (`chi2` and `f_classif`) include a `p_value` attribute that gives you the direct p-values according to the hypothesis that they are testing. The `mutual_info_classif` is not a statistical test so the output of the function is used directly as the score.

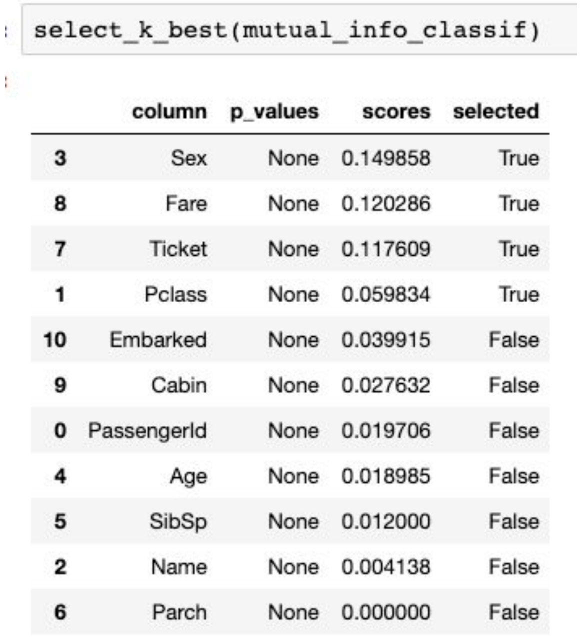
It's outside the scope of this LU to go into depth how each of these methods work, especially since there are so many of them available for all of your different models. Before choosing which to use, you should have a cursory understanding of your dataset and have a strong justification for using it BEFORE you apply the method and not after.

SelectKBest examples



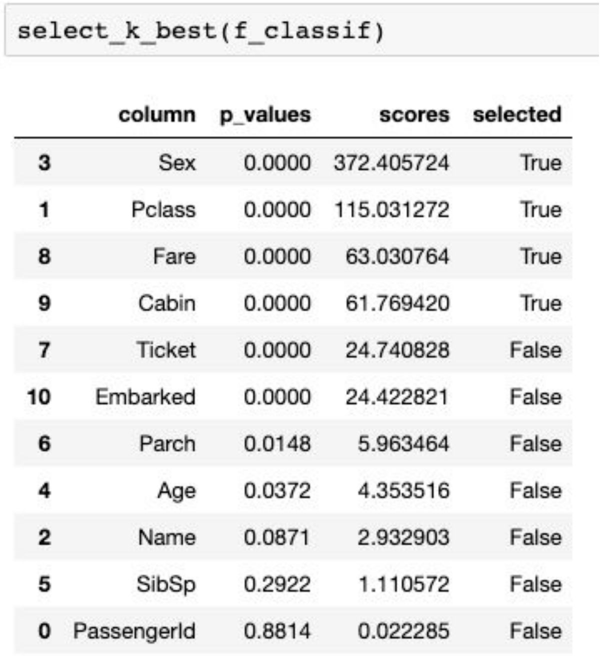
```
select_k_best(chi2)
```

	column	p_values	scores	selected
8	Fare	0.0000	4518.319091	True
7	Ticket	0.0000	2871.655466	True
9	Cabin	0.0000	573.925858	True
2	Name	0.0000	435.568915	True
3	Sex	0.0000	92.702447	False
1	Pclass	0.0000	30.873699	False
4	Age	0.0000	24.687926	False
6	Parch	0.0015	10.097499	False
10	Embarked	0.0018	9.755456	False
0	PassengerId	0.0684	3.320379	False
5	SibSp	0.1081	2.581865	False



```
select_k_best(mutual_info_classif)
```

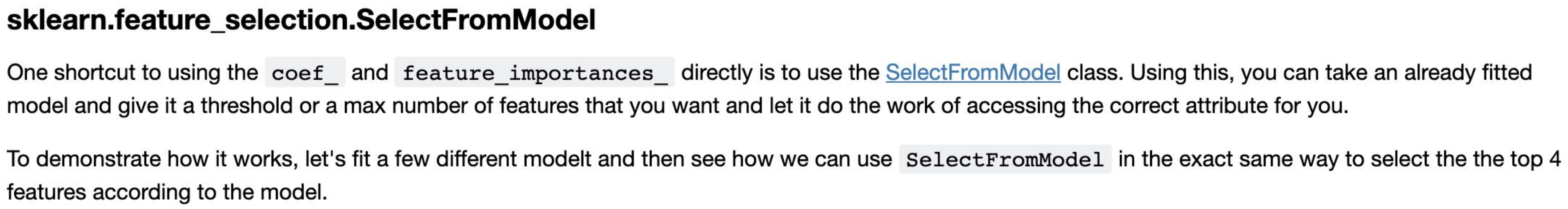
	column	p_values	scores	selected
3	Sex	None	0.149858	True
8	Fare	None	0.120286	True
7	Ticket	None	0.117609	True
1	Pclass	None	0.059834	True
10	Embarked	None	0.039915	False
9	Cabin	None	0.027632	False
0	PassengerId	None	0.019706	False
4	Age	None	0.018985	False
5	SibSp	None	0.012000	False
2	Name	None	0.004138	False
6	Parch	None	0.000000	False



```
select_k_best(f_classif)
```

	column	p_values	scores	selected
3	Sex	0.0000	372.405724	True
1	Pclass	0.0000	115.031272	True
8	Fare	0.0000	63.030764	True
9	Cabin	0.0000	61.769420	True
7	Ticket	0.0000	24.740828	False
10	Embarked	0.0000	24.422821	False
6	Parch	0.0148	5.963464	False
4	Age	0.0372	4.353516	False
2	Name	0.0871	2.932903	False
5	SibSp	0.2922	1.110572	False
0	PassengerId	0.8814	0.022285	False

SelectFromModel



sklearn.feature_selection.SelectFromModel

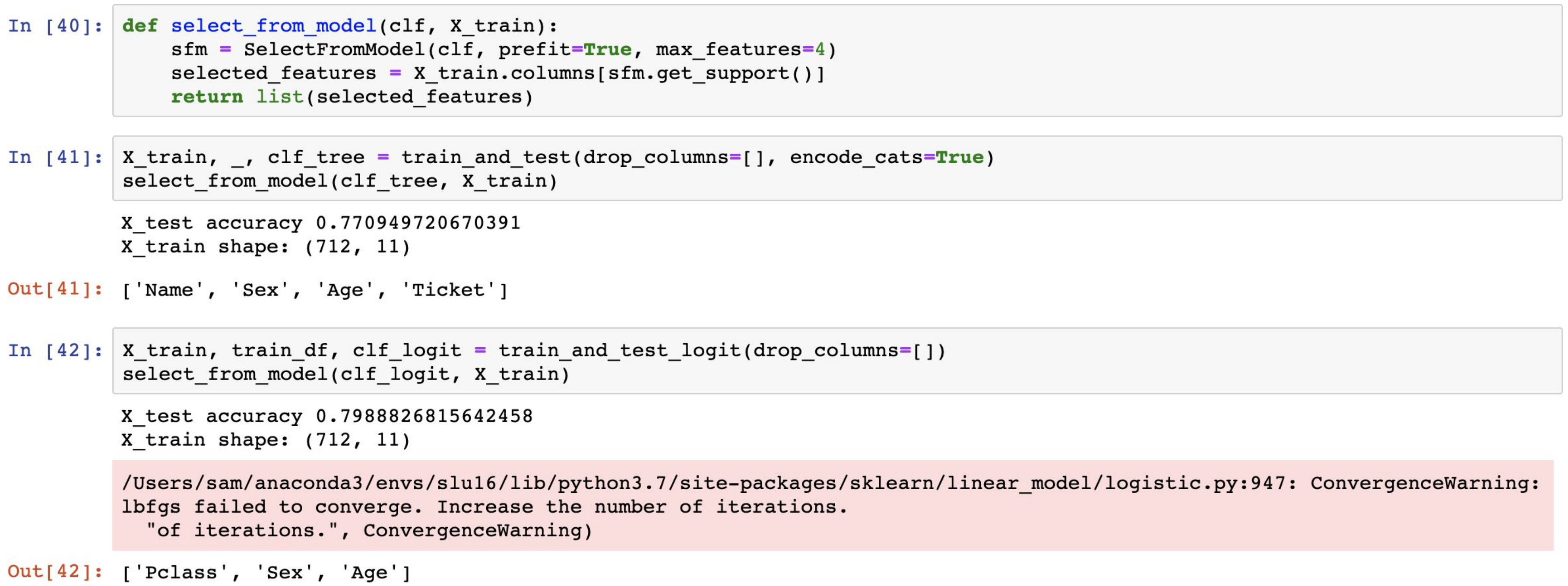
One shortcut to using the `coef_` and `feature_importances_` directly is to use the SelectFromModel class. Using this, you can take an already fitted model and give it a threshold or a max number of features that you want and let it do the work of accessing the correct attribute for you.

To demonstrate how it works, let's fit a few different modelt and then see how we can use `SelectFromModel` in the exact same way to select the the top 4 features according to the model.

SelectFromModel Usage

Once again, be careful with using these tools without understanding them or the data! As you can see from these examples, there are a lot of ways that this can go wrong if you blindly use them without thinking!

SelectFromModel Examples



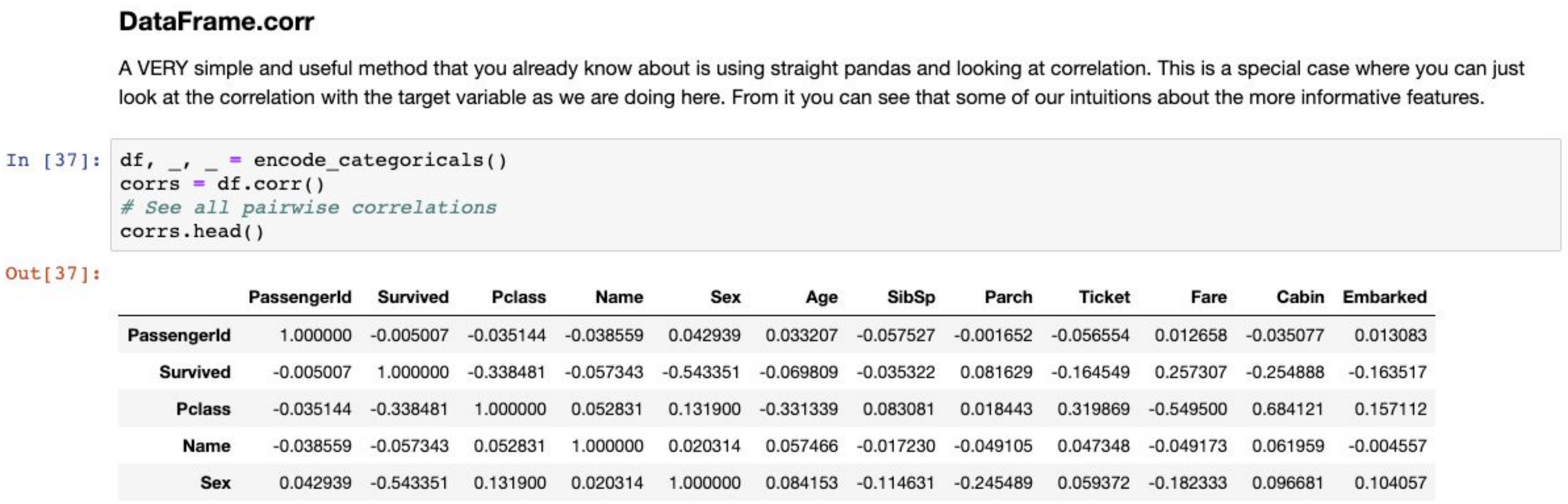
```
In [40]: def select_from_model(clf, X_train):
             sfm = SelectFromModel(clf, prefit=True, max_features=4)
             selected_features = X_train.columns[sfm.get_support()]
             return list(selected_features)
```

```
In [41]: X_train, _, clf_tree = train_and_test(drop_columns=[], encode_cats=True)
         select_from_model(clf_tree, X_train)

         X_test accuracy 0.770949720670391
         X_train shape: (712, 11)

Out[41]: ['Name', 'Sex', 'Age', 'Ticket']
```

```
In [42]: X_train, train_df, clf_logit = train_and_test_logit(drop_columns=[])
         select_from_model(clf_logit, X_train)

         X_test accuracy 0.7988826815642458
         X_train shape: (712, 11)

         /Users/sam/anaconda3/envs/slu16/lib/python3.7/site-packages/sklearn/linear_model/logistic.py:947: ConvergenceWarning:
         lbfgs failed to converge. Increase the number of iterations.
           "of iterations.", ConvergenceWarning)

Out[42]: ['Pclass', 'Sex', 'Age']
```

$$\begin{tikzcd}\text{Univariate Selection} \arrow[r] & \text{Feature Importances} \arrow[r] & \boxed{\textbf{Correlation}} \arrow[r] & \text{Learning Curves}\end{tikzcd}$$

Correlation



DataFrame.corr

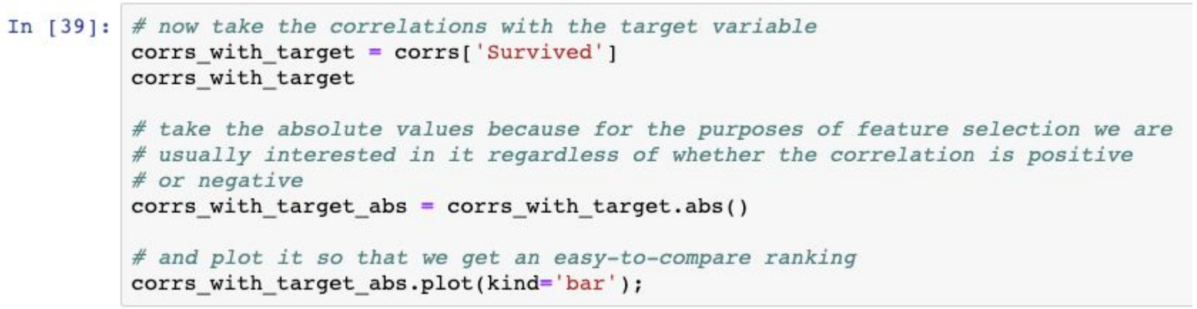
A VERY simple and useful method that you already know about is using straight pandas and looking at correlation. This is a special case where you can just look at the correlation with the target variable as we are doing here. From it you can see that some of our intuitions about the more informative features.

In [37]:
```
df, _, _ = encode_categoricals()
corrs = df.corr()
# See all pairwise correlations
corrs.head()
```

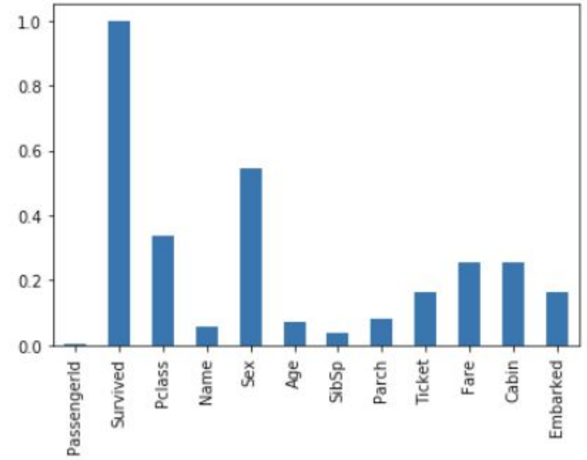
Out[37]:

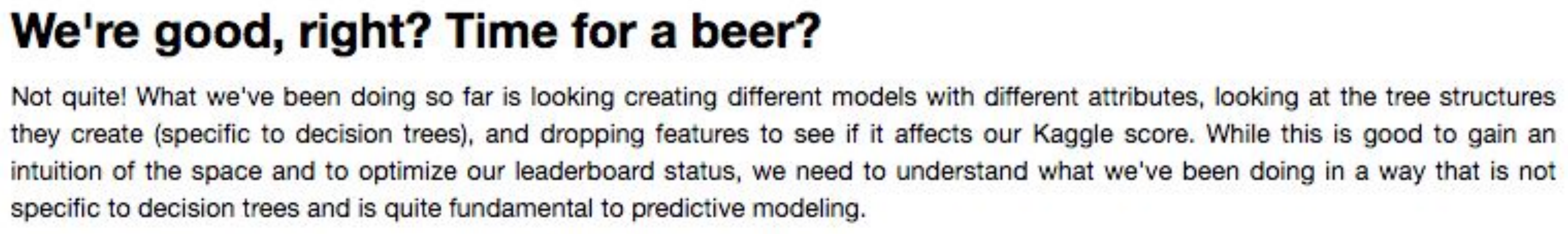
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
PassengerId	1.000000	-0.005007	-0.035144	-0.038559	0.042939	0.033207	-0.057527	-0.001652	-0.056554	0.012658	-0.035077	0.013083
Survived	-0.005007	1.000000	-0.338481	-0.057343	-0.543351	-0.069809	-0.035322	0.081629	-0.164549	0.257307	-0.254888	-0.163517
Pclass	-0.035144	-0.338481	1.000000	0.052831	0.131900	-0.331339	0.083081	0.018443	0.319869	-0.549500	0.684121	0.157112
Name	-0.038559	-0.057343	0.052831	1.000000	0.020314	0.057466	-0.017230	-0.049105	0.047348	-0.049173	0.061959	-0.004557
Sex	0.042939	-0.543351	0.131900	0.020314	1.000000	0.084153	-0.114631	-0.245489	0.059372	-0.182333	0.096681	0.104057

Correlation w/ target variable



```
In [39]: # now take the correlations with the target variable
         corrs_with_target = corrs['Survived']
         corrs_with_target

         # take the absolute values because for the purposes of feature selection we are
         # usually interested in it regardless of whether the correlation is positive
         # or negative
         corrs_with_target_abs = corrs_with_target.abs()

         # and plot it so that we get an easy-to-compare ranking
         corrs_with_target_abs.plot(kind='bar');
```

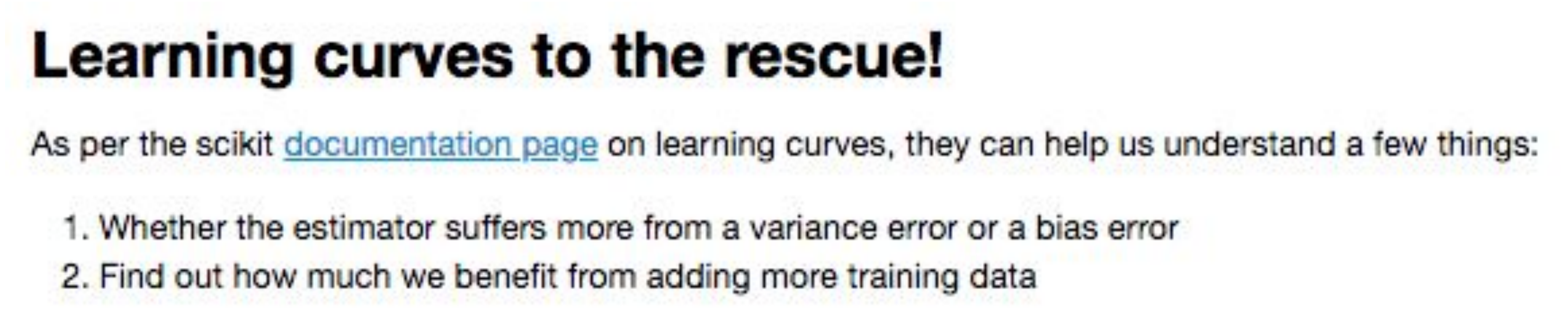




We're good, right? Time for a beer?

Not quite! What we've been doing so far is looking creating different models with different attributes, looking at the tree structures they create (specific to decision trees), and dropping features to see if it affects our Kaggle score. While this is good to gain an intuition of the space and to optimize our leaderboard status, we need to understand what we've been doing in a way that is not specific to decision trees and is quite fundamental to predictive modeling.

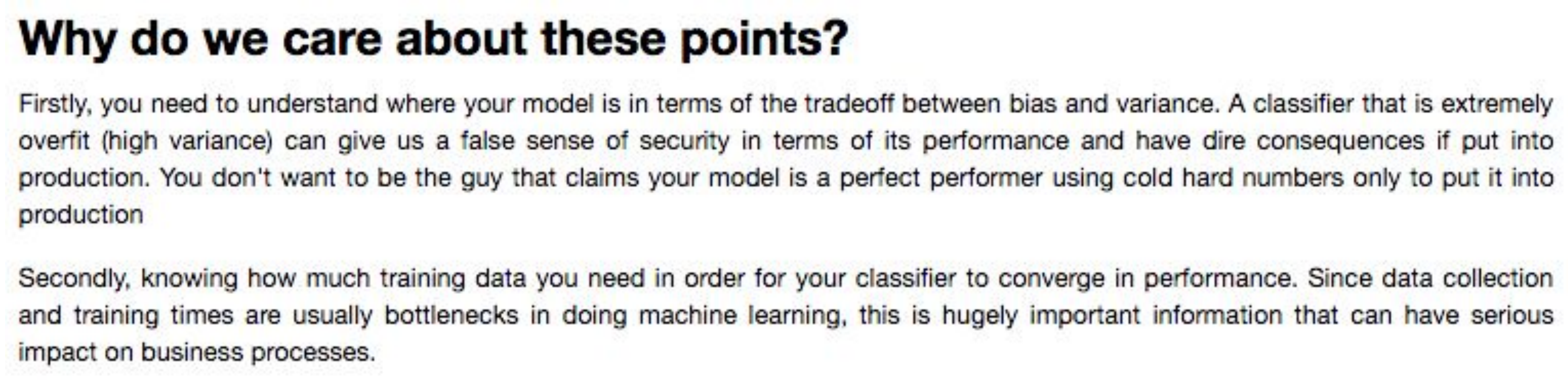
Univariate Selection ⇨ Feature Importances ⇨ Correlation ⇨ **Learning Curves**



Learning curves to the rescue!

As per the scikit documentation page on learning curves, they can help us understand a few things:

1. Whether the estimator suffers more from a variance error or a bias error
2. Find out how much we benefit from adding more training data



Why do we care about these points?

Firstly, you need to understand where your model is in terms of the tradeoff between bias and variance. A classifier that is extremely overfit (high variance) can give us a false sense of security in terms of its performance and have dire consequences if put into production. You don't want to be the guy that claims your model is a perfect performer using cold hard numbers only to put it into production

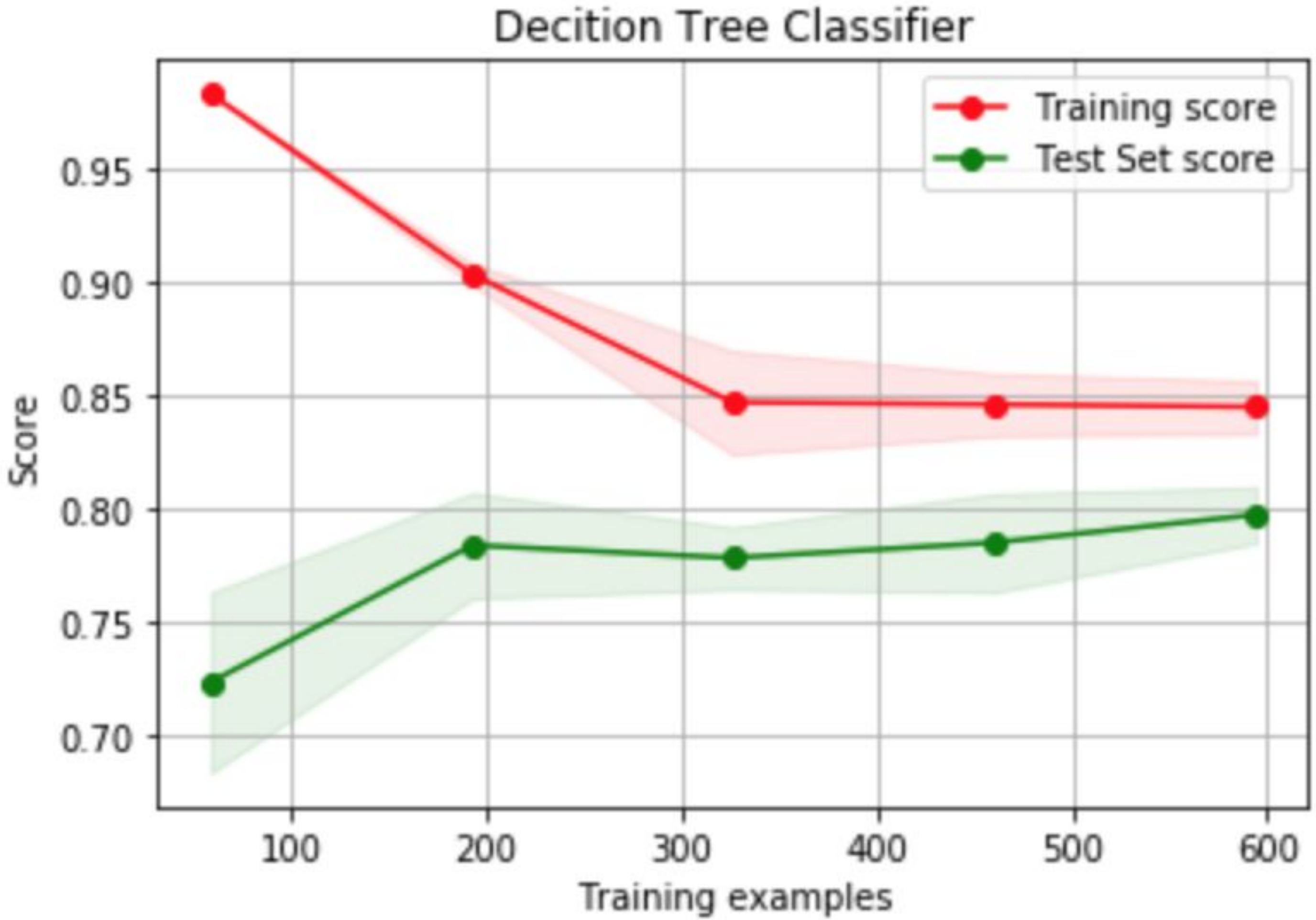
Secondly, knowing how much training data you need in order for your classifier to converge in performance. Since data collection and training times are usually bottlenecks in doing machine learning, this is hugely important information that can have serious impact on business processes.

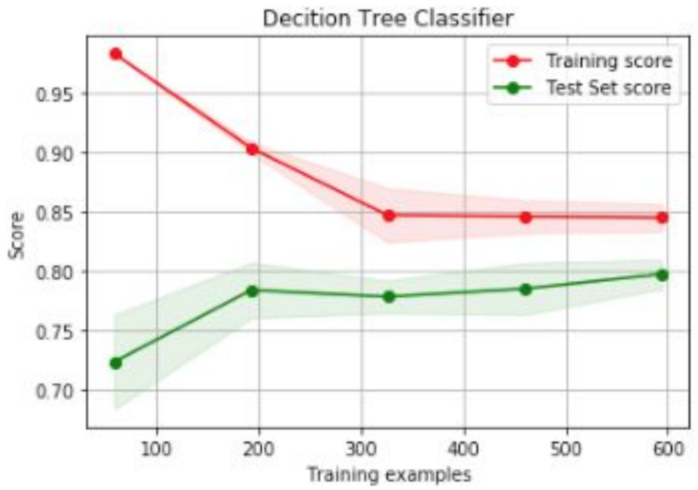
Let’s look at some learning curves!

Scenario number 1

First, let's look at the learning curve of the last model that we ended up with.

```
drop_columns = [
    'PassengerId',
    'Name',
    'Ticket',
    'Cabin',
    'Embarked',
    'Parch',
    'SibSp'
]
# Create X, y from the original dataset
_, X, y = read_and_get_dummies(drop_columns)
plot_learning_curve(DecisionTreeClassifier(random_state=1, max_depth=5), 'Decition Tree Classifie
r', X, y);
```


Decition Tree Classifier
Training score
Test Set score
Score
0.95
0.90
0.85
0.80
0.75
0.70
100
200
300
400
500
600
Training examples



Alright, let's take a few conclusions from these learning curves

1. With a low number of samples, the classifier scores VERY high on the training set and has the lowest score on the test set. This makes some sense because it has so little training data that all it can do is build a classifier that predicts the training set and has no chance to generalize. This is very overfit because it doesn't have enough training data.
2. When we reach about 300 samples, we start to reach a steady state between the training and test score. That means that for this particular model on this particular dataset, it doesn't seem to benefit much from more data.
    - This is actually super important in case you have some business requirements around how much data you can reasonably collect for user experience or regulatory reasons for example.

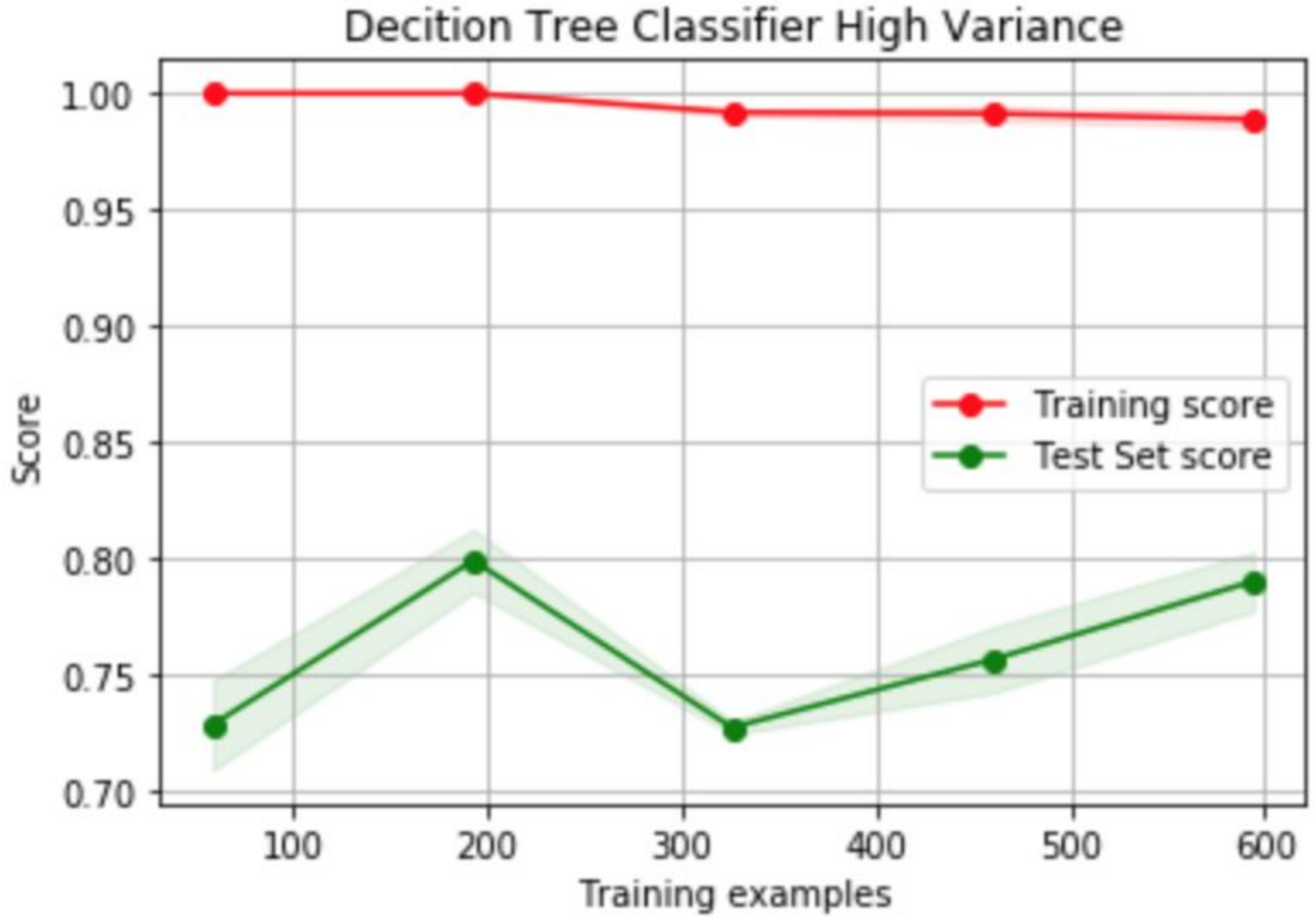
This learning curve can be considered a healthy one. It overfits on a small amount of data and can't generalize which is expected but when you give it more data, the training and test scores start to converge though the training score is always higher.

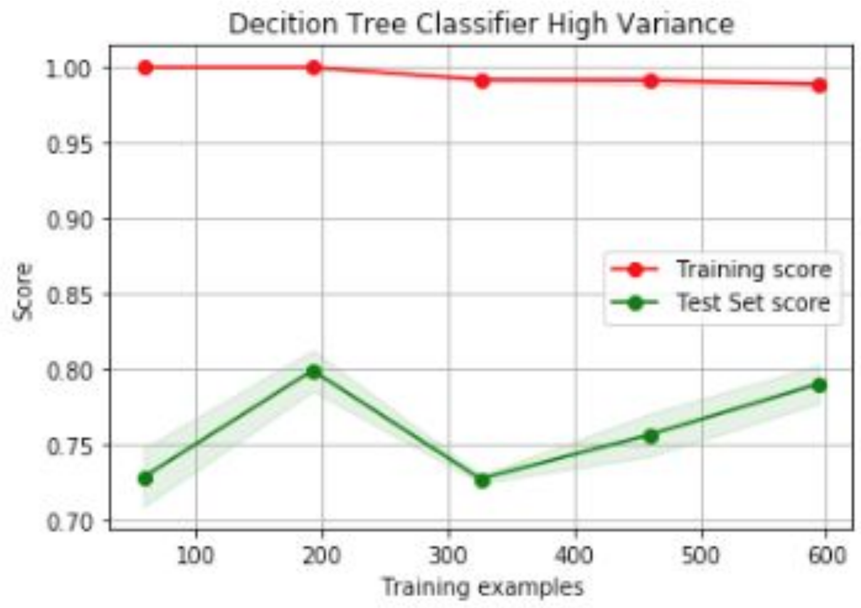
Scenario number 2

Now let's take a look at one of our more over fit models that we built today.

This is the scenario in which we experienced the performance drop on the Kaggle leaderboard from 77 down to 73 where we got rid of 3 columns (one being ticket) and didn't restrict the tree depth.

```
drop_columns = [
    'PassengerId',
    'Name',
    'Ticket'
]

_, X, y = read_and_get_dummies(drop_columns)

plot_learning_curve(
    X=X,
    y=y,
    estimator=DecisionTreeClassifier(random_state=1),
    title='Decition Tree Classifier High Variance',
);
```


Decition Tree Classifier High Variance
Score
1.00
0.95
0.90
0.85
0.80
0.75
0.70
100
200
300
400
500
600
Training examples
Training score
Test Set score



Conclusions on this one:

1. Here is a model with VERY high variance - it is almost perfect at predicting the training set (a huge sign of overfitting)
2. The amount of training data doesn't seem to have any effect on the performance. It actually lowered the test set performance significantly when going from 200->300 samples while staying almost perfect on the test set.

This is a scarily overfitted model - don't be the data scientist that puts this model into production!

3. Exercise

Notes for Exercises

- Be sure to check the end of the notebook for the low levels of how to use the learning_curve function from sklearn
- With this you'll learn the basics of how to draw the learning curves that we just looked at
- You'll need it for the exercises!

4. Recap

Overview

- Univariate Selection
- Feature Importances
- Correlation
- Learning Curves

jah tah!

Get started on the exercises :-)