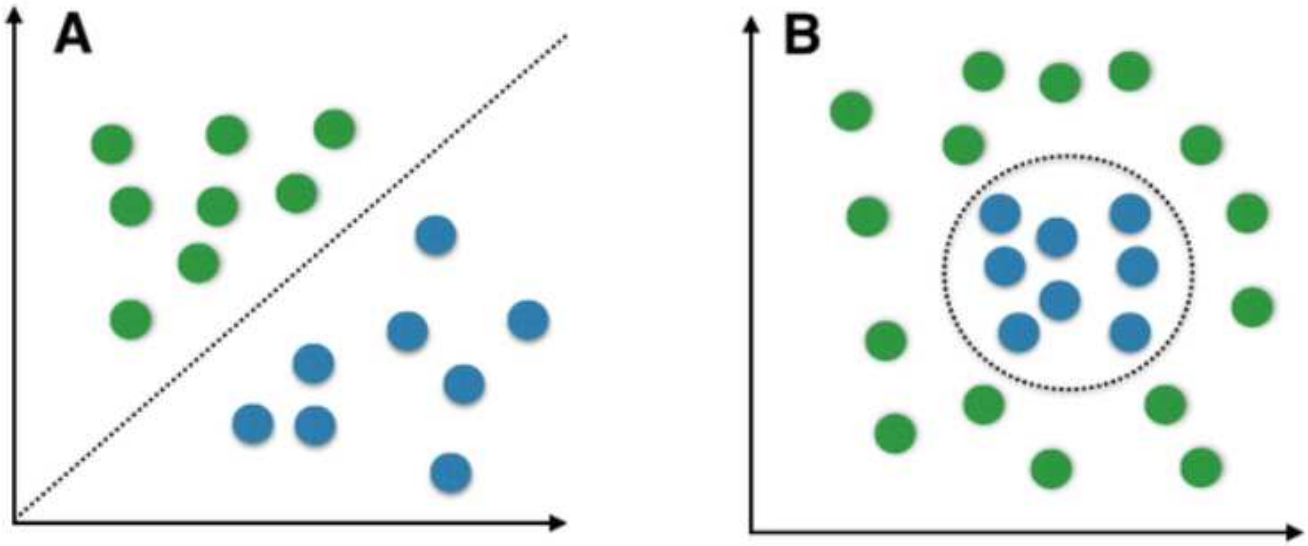
Training and Testing Machine Learning Models

A framework to working with ML models.

Alex Strebeck Jun 5, 2018 · 13 min read



Intro

The intent of this tutorial is to get you (maybe a beginner, maybe not) up and running with machine learning models. We will cover the basics from train/test splitting your data to testing and saving your model for deployment. If you follow along you will be able to deploy many of the machine learning models available from the Scikit-learn library with only a minor amount of tweaks! If your a beginner or intermediate learner you probably can learn something here too. Before we start you

will need to find and clean a data set of your choosing. I will be using the titanic data set as there is a lot of information already out there about cleaning this data.

Using Scikit-Learn Train/Test Split

A common problem in machine learning algorithms is their tendency to "memorize" the data they have been trained on. In data science terms this is called over-fitting the data and can make your model look great on the training data but in actuality it has no ability to generalize on new data that is given. To gain perspective into how the model is actually doing we use a train/test split. Simply put we just select a certain percentage of our data and withhold it from the "eyes" of the machine learning algorithm. In this way, at the end of training there is still a chunk of data that we can test the model on and see how it performs in comparison to the training data. Often times when you compare the score of the test and training data the training data will have a lower score but in good models the test score should be close by. Lets test this quickly with the logistic regression classifier from Scikit-Learn.

Lets first import the train_test_split model from the Scikit library. Then we assign the "features" or columns that we want to use for prediction to "x" and our "labels" or column that we are predicting to "y". Next we use both x and y as our data for the train_test_split function also specifying the test size that we want. Additionally we can assign a random_state which is optional, and does not effect the quality of our data.

```
from sklearn.model_selection import train_test_split

x = titanic_cleaned[["pclass", "sex", "age", "sibsp", "parch",\
"fare","adult_male", "embark_town"]]
y = titanic_cleaned["survived"]
```

```
x_train, x_test, y_train, y_test = train_test_split(x, y,\
test_size = 0.33, random_state=42)
```

Next we plug our cleaned and split data into our logistic regression algorithm, which we have to import. We first train or "fit" our model on the training split. Then to get a proper score of the model we score it based on the test data. You can also score the model on the training data which can give us an idea of the score points lost between the training and test runs. This can provide hints into how much the model is over-fitting.

```
from sklearn.linear_model import LogisticRegression

regression = LogisticRegression(max_iter = 1000)
regression.fit(x_train, y_train )

print("Train Accuracy:",regression.score(x_train, y_train))
print("Test Accuracy:",regression.score(x_test, y_test))
```

Now that we have trained our model lets see how it did on the data.

Full Data after clean - 784 rows	Score
Training data - 525 rows	80.5 %
Testing data – 259 rows	79.9 %

As you can see from the table, the scores are very close which indicates that we avoided over-fitting. I should mention that this is a good indication that we have not over-fit the model, however it is not the end all be all. Next we'll discuss another method to prevent over-fitting of our data and hopefully improve our ability to generalize over new data.

Using Scikit-Learn cross_validation library

In addition to a train/test split, cross validation can be very useful in understanding the quality of your model's predictions. Hidden in every set of data is an inherent amount of bias towards a certain prediction outcome. A way to think about it is that when you select a certain amount of points to train your model on, statistically speaking you will never sample your data in such a way as to be perfectly "fair" to each outcome and the features that perfectly let you predict on that data. In this way you are effectively including a bias to predict certain classes more frequently than others. As you may have guessed by now, cross validation can help combat this and inform us about the true state of our model.

In this article we will specifically be using Kfold method of the Scikit-Learn library. What Kfold cross validation does is shift the data in which we are testing and training on. The table below show a visual for of how this works. When you enact Kfold in your model what it essential does is divide up your data in "K" number of folds or segments, the figure below shows $K = 4$. One section is used for testing while the rest of the segments are used for training. It then calculates a score for each time the model was trained and tested. This is done so that each segment is tested on exactly once. What this allows is for a true estimate of how the model is actually performing, in this fashion we can remove a lot of bias (not all) from test score since it has trained on all of the data throughout the trials.

Kfold Visualization

This can be used with the following code in python. X_train and Y_train are defined as before. My personal preference is to still leave out a test data set as to get a score on truly unseen data.

```
#Imports
import numpy as np
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn import model_selection


#seed for random state and splits - number of Kfolds
random_seed = 1
splits = 5


#Scikit Kfold model call
```

```
kfold = model_selection.KFold(n_splits=splits,\
random_state=random_seed)


#Logistic Regression Model
model = LogisticRegression()


#results set equal to model with kfolds
results = model_selection.cross_val_score(model, x_train,\
y_train,cv=kfold)


#printing the averaged score over the 5 Kfolds
print("Accuracy: %.3f%% (%.3f%%)" % (results.mean()*100.0,
results.std()*100.0))
```

Choose Model

Now that we have our data split and we know a basic way of testing our model with Kfolds and on the test set, we need to figure out where to start in finding the best model for out application. There are many ways to go about this, however a rather simple and good first pass attempt is to just train a few different models on the same data and see what score they each achieve “out of the box”. As we will see later, we can take the top two best performing models and tune them each individually to get even better results. In order to score each of the model we will be using a indicator called Area under the Curve which will be discussed later. The code below shows each of the models I used in this example. If you are new to ML it is not important just yet to understand how the model works under the hood but more importantly to understand how they initially score. This will allow us to make a decision on the top *n* number of models we want to use, for now we will pick the top 2. Below are the results from 5 different Scikit-learn models.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
```

```
model = LogisticRegression()
cross_val = KFold(n_splits=3, random_state=42)
scores = cross_val_score(model, x_train, y_train, cv=cross_val,
scoring='roc_auc')
print("Mean AUC Score - Logistic Regression: ", scores.mean()
```

Mean AUC Score — Logistic Regression: 0.845

```
from sklearn.ensemble import RandomForestClassifier

model = RandomForestClassifier()

kfold = KFold(n_splits=3, random_state=42)
scores = cross_val_score(model, x_train, y_train, cv=kfold,
scoring='roc_auc')
print("Mean AUC Score - Random Forest: ", scores.mean())
```

Mean AUC Score — Random Forest: 0.838

```
### Decision Tree
from sklearn.tree import DecisionTreeClassifier
model2 = DecisionTreeClassifier()
cross_val = KFold(n_splits=3, random_state=42)
scores = cross_val_score(model2, x_train, y_train, cv=cross_val,
scoring='roc_auc')
print("Mean AUC Score - Decision Tree: ", scores.mean())
```

Mean AUC Score — Decision Tree: 0.756

```
### Naive Bayes
from sklearn.naive_bayes import GaussianNB
model4 = GaussianNB()
cross_val = KFold(n_splits=3, random_state=42)
scores = cross_val_score(model4, x_train, y_train, cv=cross_val,
scoring='roc_auc')
print("Mean AUC Score - Gaussian Naive Bayes: ", scores.mean())
```

Mean AUC Score — Gaussian Naive Bayes: 0.833

```
### K-Nearest Neighbors
from sklearn.neighbors import KNeighborsClassifier
model5 = KNeighborsClassifier()
cross_val = KFold(n_splits=3, random_state=42)
scores = cross_val_score(model5, x_train, y_train, cv=cross_val,
scoring='roc_auc')
print("Mean AUC Score - K-Nearest Neighbors: ", scores.mean())
```

Mean AUC Score — K-Nearest Neighbors: 0.712

Results

From the 5 scores above the top 2 performers were Logistic Regression and Random Forest. We will move ahead and fine tune them.

Identify hyper parameters for tuning

Once we figured out which models we want to move forward with, the fast easy work is over with. At this point we need to identify which parameters can be tuned in each of the models we have chosen.

Lets have a look at the logistic regression so that we can get an understanding of what happens when we change hyper parameters.

From the Scikit docs, we can glean some information into which parameters we can change for this model. After giving it a good look, for our example here, I am going to stick with penalty and C as tune-able parameters. Lets see how our score changes if we change these parameters around.

Logistic Regression AUC Score: **0.5397**, C of **0.001** and a **l1** penalty

Logistic Regression AUC Score: **0.8487** C of **1** and a **l2** penalty

Logistic Regression AUC Score: **0.8535** C of **10** and a **l2** penalty

As you can see just from changing these 2 parameters we can greatly effect the quality of our model. Something to keep in mind however, is that while this can greatly impact our model score, it can take a lot of manual work just trying to squeeze a little extra out of our model. With that being said, doing this manually can be a huge waste of time but it can definitely pay off, so what should we do?

Automate it!

Using grid search for exploring hyper parameters

As we saw in the last section , hyper parameter tuning can have a significant effect on your model and its ability to make accurate predictions on data. Hyper parameter tuning is not really a hard task to do, it just takes a lot of time. This just happens to be a perfect job for a computer to do, why should we sit around plugging numbers into a program when we can just have a computer try a bunch for us?

This fact is the very idea behind a form of hyper parameter optimization

called **grid search**. Once you have identified the parameters that you think have the greatest effect on the score of your model you'll want to make sure they get optimized. Effectively, grid search takes in a list of the proposed parameters you want to optimize and the values at which you want to test them and greats a grid from these, like in the figure below.

Parameters	Values													
C	0.001	0.01	0.1	1	10	100	1000	0.001	0.01	0.1	1	10	100	1000
Penalty	l1	l1	l1	l1	l1	l1	l1	l2	l2	l2	l2	l2	l2	l2

Grid Search Values for Logistic Regression

What this figure represents is the "grid" of values chosen, for the optimization of our logistic regression model, this is also referred to as the parameter space. For each of these sets of parameters the grid search will plug these into the model and fit the model to the training data. It also saves the score for each of these. Once complete, the grid search algorithm knows which set of parameters produced the best score, which in turn is the optimized state of the model. This can be done several times modifying the scalar valued parameters to a range close to that of the previous optimized parameters. In other words, the grid search will only find the best combo for the values entered and as we gain information about which values produce good results we can then refine the values used to hone in on the true optimized parameters. The code snippet below shows how to implement this on our logistic regression model from before.

```
from sklearn.model_selection import GridSearchCV

tuned_parameters = [{'C': np.linspace(.0001, 1000, 200) ,
'penalty': [ "l1", "l2"]}]

clf = GridSearchCV(regression, tuned_parameters, cv=cross_val,
scoring= 'roc_auc')
```

```
clf.fit(x_train, y_train)

print("Best parameters set found on development set:")
print()
print(clf.best_params_)

print("\nOptimized model achieved an ROC of:" \
,round(clf.score(x_train, y_train), 4))
```

Best parameters set found on development set:

{'C': 10.0503, 'penalty': 'l2'}

Optimized model achieved an ROC of: ***0.8615***

After the optimization has complete we can then score the proposed model to see if it really does produced a better result. In this case it does in fact boost our results by almost an entire point!

** one caveat that should be mentioned here is that although this method is automated, it can still take a long time to resolve the optimized model depending on the number of parameters entered in. In this case other optimization methods should be researched and used.

Compute AUC for a set of results

This whole time we have been evaluating our models based on some measure called the Area Under the Curve (AUC), but what is it?

Before we describe the AUC we must first get a couple of other terms out of the way, namely the receiver operating characteristic curve (ROC), true positive, false positive and the threshold value. lets start with the latter three and work our way up to ROC.

Whenever we train a classification model its job is to tell us which category a data point belongs to (this is also how we score it). The way that it decides this is based on its decision threshold value, or the probability after which a data point is classified as a positive match. In other words if we input a data point and the model says that the probability of that being a survivor (titanic data set) is 0.68 and our threshold is 0.5 then the model predicts this input as a 1 or a survivor . When we do this for a whole set of data points the model, as we have seen already, wont predict every input perfectly.

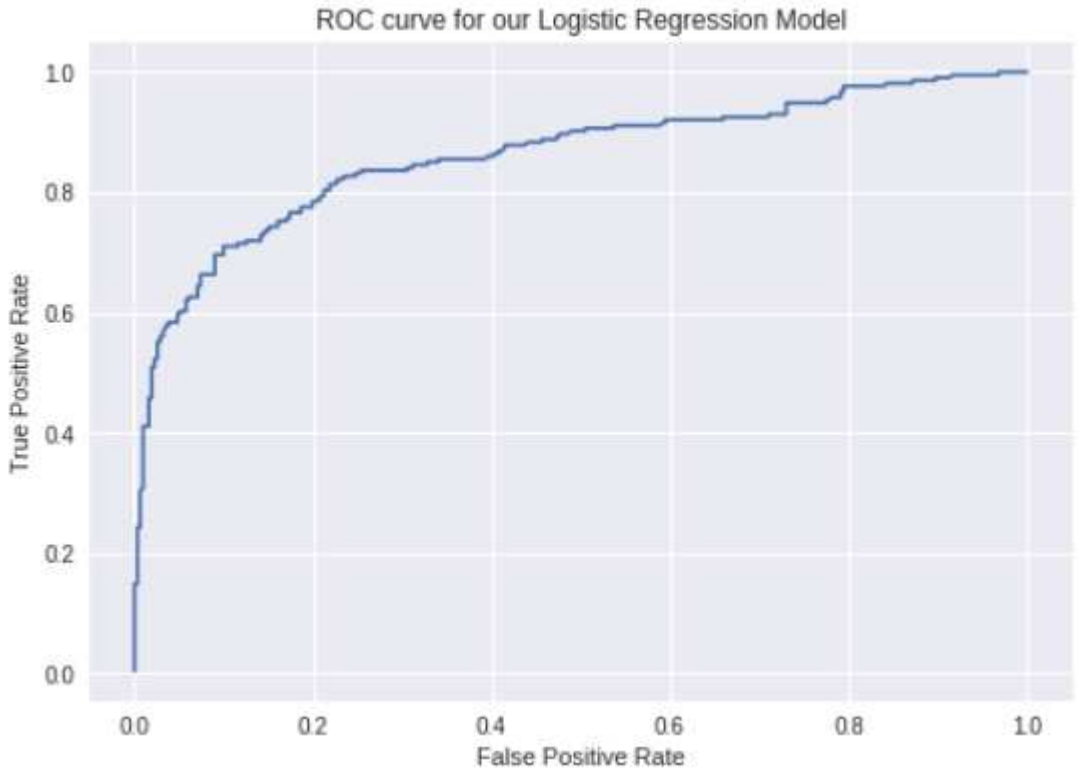
This brings us to true positives and false positives. When a model predicts a passenger surviving and the true answer is also that the passenger survived, this is said to be a true positive. This is what we want out of model. In the opposite case if it predicts a passenger surviving and the true answer is that the passenger did not survive, this is said to be a false positive. This is what we do not want. In this case the model has predicted a class that does not fit that input, this is were threshold comes in.

As the model “operator”, we could change the threshold over which the model makes a true prediction on input data. For example, instead of the 0.5 threshold we mentioned before, we could have it predict true after only having a probability of 0.4. How would this effect our outcomes? In this case since we lowered the threshold, there is a higher number of survivors predicted since the criteria (threshold value) is lower. In other words the model does not have to have as high of probability for a certain input for it to be predicted as a survivor. In this case though, since more positive (survivor) predictions are being made, we can reason that some of those predictions are wrong and result in predicting a survivor when it should not have been
(i.e. false positive). In the same vein our true positive rate also goes up at

the same time. This changing of true and false positive rates is what make an ROC curve.

As we change our threshold values lower, generally speaking both true and false positive rates will go up, but what we want is that our true positive rate goes up while the false positive goes down. This would be the ideal model. In reality we cannot produce a model thus far that is 100% right 100% of the time on every data set. In our reality ROC's tend to look like the curve below.

```
from sklearn.model_selection import GridSearchCV
from sklearn import metrics
import matplotlib.pyplot as plt


tuned_parameters = [{'C': np.linspace(.0001, 1000, 200) ,\
'penalty': [ "l1", "l2"]}]


clf = GridSearchCV(regression, tuned_parameters, cv=cross_val,\
scoring= 'roc_auc')


clf.fit(x_train, y_train)


print("Best parameters set found on development set:")
print()
print(clf.best_params_)


print("\nOptimized model achieved an ROC of: ",
round(clf.score(x_train, y_train), 4))


proba = clf.predict_proba(x_train)


# Producing the same false/true positive data via a library and
plotting it
fpr, tpr, _ = metrics.roc_curve(y_train.values,proba[:,1])
# print(fpr)
# print(tpr)
plt.plot(fpr,tpr);
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title("ROC curve for our Logistic Regression Model")
```



This is were we get our final score metric from, the Area Under the Curve (AUC). The AUC is basically the calculation of the area under the ROC, in this way we quantify how well the model does over a range of threshold values. The more area that we have under the curve, the better the model was at predicting classes and thus we have a higher rate of true positives while holding the number of false positives at a much lower rate. The big take away here is that a bigger AUC score will generally correspond to a better model.

**It should be noted that while we are trying get to a 100% true positive rate while having no false positives, we can change sometimes tolerate a higher false positive rate in order to achiever a higher true positive rate. This decision must be made with the final prediction in mind, if it is a life or death prediction this is not a good idea but if your predicting a Facebook users mood, you can tolerate more false positives.

Save model parameterization

Ahh, you have made it this far. Must mean your model is all tuned up and ready to make predictions in the real world, that's all any Data Scientist wants right?

Once a model has been tuned, we need a way of exporting that model to a place where it can be deployed in the real world. This actually is a pretty simple process, all we really need is its parameters and weights of the model that we found during our training and optimization phases.

Depending on the model you're using this can be 2 parameters or 3 parameters, all the way up to millions, fortunetly for us Scikit-learn has figured this out for us. We can simply use their library to save our file and load that file up from any place in the world as long as we access to the saved parameters. The following code snippet shows us how.


```
from sklearn.externals import joblib
joblib.dump(clf, 'filename.pkl')

model = joblib.load('filename.pkl')
```

It's that simple!

I hope you have enjoyed getting to know the basic steps of the machine learning framework. you now have the foundation to train and experiment will all types of machine learning models. Now get out there and change the world!