Pivotal (/)

Pivotal Engineering Journal (/)

Technical articles from Pivotal engineers.

Test-Driven Development for Data Science

Unravelling Test-Driven Development for Data Science.

Posted on Thu, Sep 15, 2016 by Dat Tran (https://twitter.com/datitran) , Megha Agarwal (https://twitter.com/agarwal22megha)

Categories: TDD (/categories/tdd) Data Science (/categories/data-science) Machine Learning (/categories/machine-learning) Agile (/categories/agile) Pair Programming (/categories/pair-programming)

Edit this post on GitHub. (https://github.com/pivotal-cf/blog/edit/master/content/post/test-driven-development-for-data-science.md) Tweet

Joint work by Dat Tran (https://de.linkedin.com/in/dat-tran-a1602320) (Senior Data Scientist) and Megha Agarwal (https://uk.linkedin.com/in/agarwal22megha) (Data Scientist II).

This is a follow up post on API First for Data Science (http://engineering.pivotal.io/post/api-first-for-data-science/) and Pairing for Data Scientists (http://engineering.pivotal.io/post/pairing-on-data-science/) focusing on Test-Driven Development.

Motivation

Test-Driven Development (TDD) has plethora (http://pivotal-guides.cfapps.io/craftsmanship/tdd/) of advantages. You might be wondering how is TDD relevant for data science? While building smart apps, as Data Scientists, we are really contributing in shaping the brain of the application which will drive actions in real-time. We have to ensure that the core driving component of the app is always behaving as expected, and this is where TDD comes to our rescue.

Data Science and TDD

TDD for data science (DS) can be a bit tricky than software engineering. Data science has a fair share of exploration involved, where we are trying to find features and algorithms that will contribute best to solve the problem in hand. Do we strictly test drive all our features right from the exploratory phase, when we know a lot of them might not make into production? In the initial days of exploring how TDD fits the DS space, we tried a bunch of stuff. We highlight why we started test driving our DS use case and what worked the best for us.

Now imagine you are a logistics company and this shiny little 'smart' app has figured out the best routes your drivers need to pursue the following day via some sorcery. Next day is your moment of truth! The magic better work, else you can only imagine the chaos and loss it will result in. Now what if we say we can ensure that the app will always generate the most optimised route? We suddenly have more confidence in our application. There is no secret ingredient here! We can wrap up our predictions in a test case which allows us to trust the model only when its error rate is below a certain threshold.

You might ask, why would we put a model in production which doesn't have the desired error rate at the first place? But we are dealing with real life data and predictions here, things can go haywire pretty fast. We might end up with a broken or (if you are even more lucky) no model depending on how robust the code base is. TDD can not only help us ensure that nothing went wrong while developing our model but also prompts us to think what we want to achieve and forces us to think about edge cases where things can potentially go wrong. TDD allow us to be more confident.

TDD is dead, long live TDD

So TDD saves the day, let's start test driving everything? Not quite when it comes to data science. As discussed in one of our previous blog posts (http://engineering.pivotal.io/post/api-first-for-data-science/), we have two phases in our data science engagements: *exploratory* and *production*. Test driving all the features and algorithms stringently during the exploratory phase is an overkill. We felt we were investing a lot of time in throw-away work. TDD made the exploration phase quite intensive and slowed us considerably. Eventually we found a fine balance between TDD and DS.

After the exploratory phase, once we have figured out which model and features suits the use case the best, we actually start test driving feature generation, model development and integration bits. Our model evaluation from the exploratory phase helps us set expectations around the model's performance. We leverage this information in our TDD to make sure the model is behaving as expected in production.

Here are few things to keep in mind while test driving data science use cases:

1. Start TDD once you have a fair idea of what model and features will go in production.
2. Don't test drive everything! Figure out the core functionalities of your code. For example while feature generation, if a feature is generated using simple count functionality of SQL, trust that SQL's in-build count functionality is already well tested. Writing another test around that won't add much value.
3. At Pivotal we pair programme (http://engineering.pivotal.io/post/pairing-on-data-science/). So we test drive things in a ping pong manner, i.e. one person comes up with a test case, the companion makes it pass and then the role reverses. This makes TDD interesting! :)
4. Have separate unit and integrations test suite.
5. Mock behaviours where appropriate.
6. Use an appropriate CI tool for automated testing, for example Concourse CI (https://concourse.ci/) is a good choice.
7. As a general rule of thumb, ensure your entire test suite runs within 10 min.
8. Test those bits of pipeline where data is involved. The work product of a data science use case is data.

Example

Let's demonstrate TDD for data science with a simple example. Assume we are given two features x and y for five observations and we want to assign these data points to meaningful clusters.

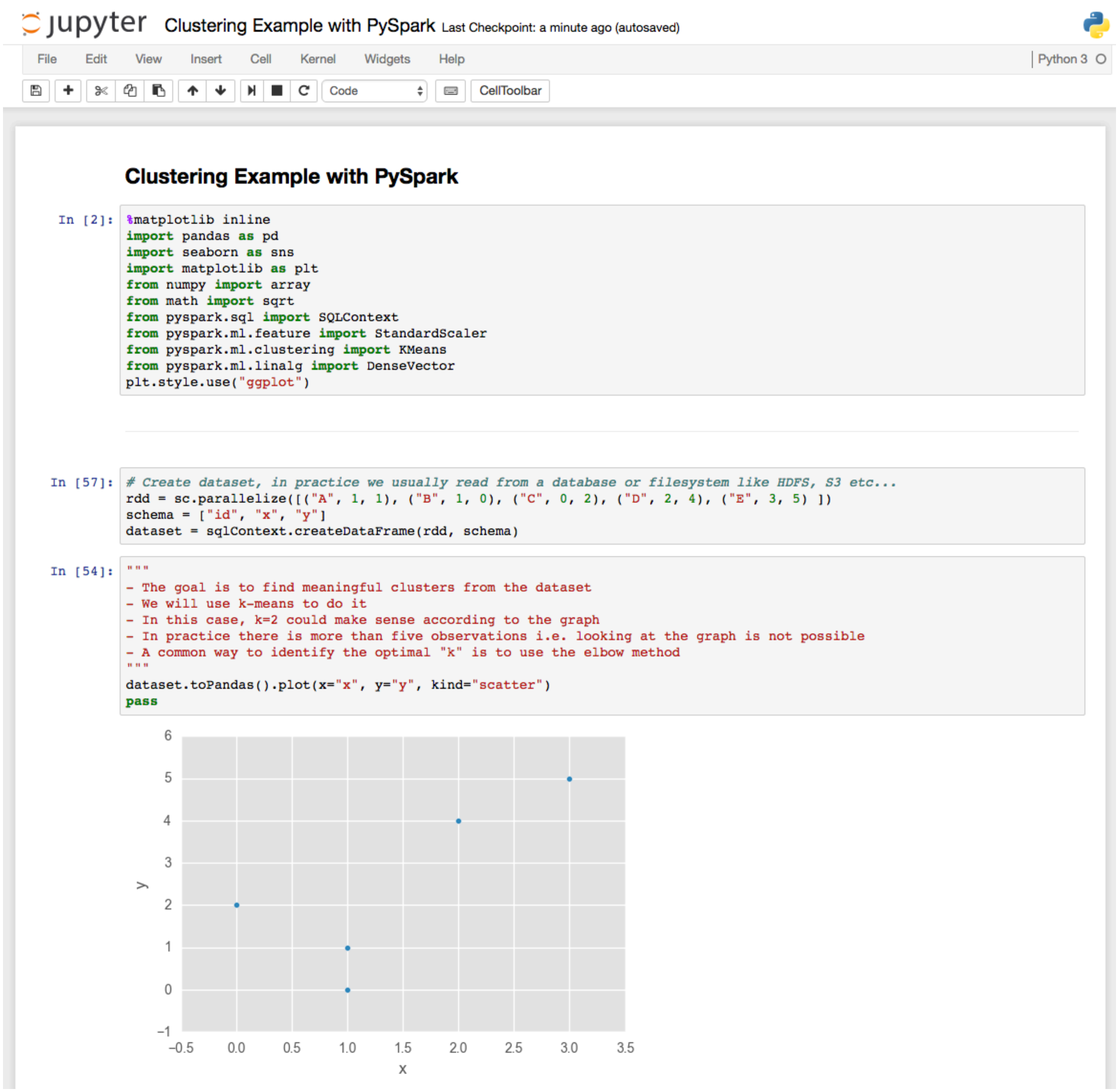
Obs	**X**	**Y**
A	1	1
B	1	0
C	0	2
D	2	4

E	3	5

This is an unsupervised problem (https://en.wikipedia.org/wiki/Unsupervised_learning) where we try to find structure in the data. We can find potential applications of clustering vividly. For example it is applied in many marketing divisions where customer segmentation is crucial to develop personalized campaigns. In real life, our data might be much larger. We've dealt with terabytes of data having several hundred features or millions of observations. In order to address such a problem, we leverage MLlib (http://spark.apache.org/docs/latest/ml-guide.html), Spark's machine learning (ML) library. Spark (http://spark.apache.org/) is handy for big data processing and supports Python, R, Scala and Java. For this example we will use PySpark (http://spark.apache.org/docs/latest/api/python/index.html), Spark's Python API.

Exploration Phase

In the exploration phase we don't test drive our code. We aim to find the right algorithm to solve our problem. We will use Jupyter notebook to play around with the data (see figure 1, the notebook can be found here (https://github.com/datitran/spark-tdd-example/blob/master/Clustering%20Example%20with%20PySpark.ipynb)). For the given dataset, k-means (https://en.wikipedia.org/wiki/K-means_clustering) can be an appropriate solution.


jupyter Clustering Example with PySpark Last Checkpoint: a minute ago (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Python 3
Code
CellToolbar

Clustering Example with PySpark

In [2]:
```
%matplotlib inline
import pandas as pd
import seaborn as sns
import matplotlib as plt
from numpy import array
from math import sqrt
from pyspark.sql import SQLContext
from pyspark.ml.feature import StandardScaler
from pyspark.ml.clustering import KMeans
from pyspark.ml.linalg import DenseVector
plt.style.use("ggplot")
```

In [57]:
```
# Create dataset, in practice we usually read from a database or filesystem like HDFS, S3 etc...
rdd = sc.parallelize([("A", 1, 1), ("B", 1, 0), ("C", 0, 2), ("D", 2, 4), ("E", 3, 5) ])
schema = ["id", "x", "y"]
dataset = sqlContext.createDataFrame(rdd, schema)
```

In [54]:
```
"""
- The goal is to find meaningful clusters from the dataset
- We will use k-means to do it
- In this case, k=2 could make sense according to the graph
- In practice there is more than five observations i.e. looking at the graph is not possible
- A common way to identify the optimal "k" is to use the elbow method
"""
dataset.toPandas().plot(x="x", y="y", kind="scatter")
pass
```

Figure 1: Jupyter notebook of our clustering example with PySpark

Figure 2 depicts that k-means did a decent job in clustering the dataset. In this case, clustering was fairly easy because of the few artificially generated data points and variables. In real life choosing the optimal *k* becomes less obvious because of more observations and/or features. Also we have to figure out which clustering algotithm (http://scikit-learn.org/stable/modules/clustering.html#clustering) is best for a given data set. In this case, k-means seems to be a good choice.

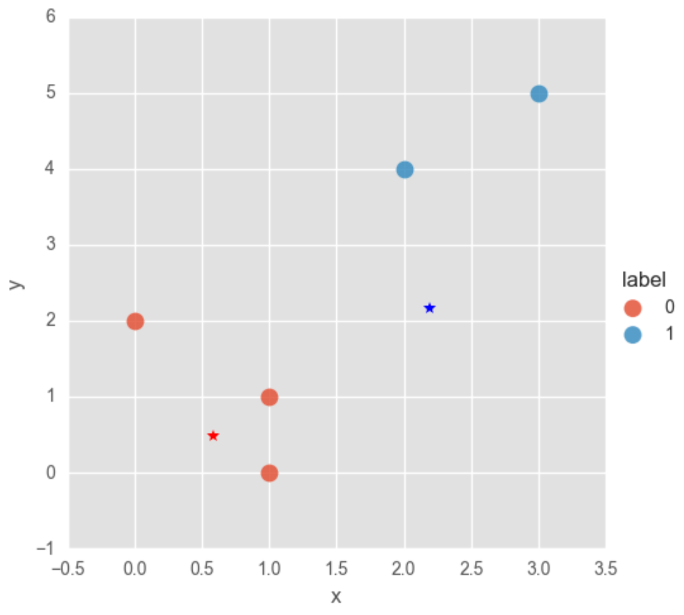


Figure 2: The data set along with the assigned labels from k-means and cluster centers

Production Phase

Now we want to operationalize our model so that we can make predictions on incoming data in real time. We want to update our model, re-calculate our cluster centers on a regular basis, to incorporate information from new data points. In this phase, we use test-driven development to ensure that our codebase is clean, robust and trustworthy.

To test our code, we use a single node Spark application. On OS X, Apache Spark can be installed using brew (http://brew.sh/):

```
brew install apache-spark
```

Next we add this to the bash profile:

```
export SPARK_HOME="/usr/local/Cellar/apache-spark/2.0.0/libexec/"
```

Tip: You might want to change Spark logger setting from `INFO` to `ERROR`

Change logging settings:

- *`cd /usr/local/Cellar/apache-spark/2.0.0/libexec/conf`*
- *`cp log4j.properties.template log4j.properties`*
- *Set info to error: `log4j.rootCategory=ERROR, console`*

This is because the default Spark logging is quite verbose and you might want to tune it down.

We add our test case to `test_clustering.py` script and implement clustering in `clustering.py` script. Let's start test driving our code:

```
import os
import sys
import unittest

try:
    # Append PySpark to PYTHONPATH / Spark 2.0.0
    sys.path.append(os.path.join(os.environ["SPARK_HOME"], "python"))
    sys.path.append(os.path.join(os.environ["SPARK_HOME"], "python", "l
ib",
                                 "py4j-0.10.1-src.zip"))
except KeyError as e:
    print("SPARK_HOME is not set", e)
    sys.exit(1)

try:
    # Import PySpark modules here
    from pyspark import SparkConf
    from pyspark.sql import SparkSession
except ImportError as e:
    print("Can not import Spark modules", e)
    sys.exit(1)
```

We added the `SPARK_HOME` to our `PYTHONPATH`, so that it recognizes PySpark. We import SparkSession which is now the new entry point into Spark since version `2.0.0`.

Then we define a single node Spark application in the test `setUp` and stop it in the `tearDown`. This means that every time we run a test, a new Spark application is created and stopped after the test is completed.

```
# Import script modules here
import clustering

class ClusteringTest(unittest.TestCase):

    def setUp(self):
        """Create a single node Spark application."""
        conf = SparkConf()
        conf.set("spark.executor.memory", "1g")
        conf.set("spark.cores.max", "1")
        conf.set("spark.app.name", "nosetest")
        SparkSession._instantiatedContext = None
        self.spark = SparkSession.builder.config(conf=conf).getOrCreate
()
        self.sc = self.spark.sparkContext
        self.mock_df = self.mock_data()

    def tearDown(self):
        """Stop the SparkContext."""
        self.sc.stop()
        self.spark.stop()
```

Next step is to mock the data. In our case since the data is small, we can use it straight ahead. In practice we could have millions of observations or variables and we need to take a subset of the real data. The amount and kind of data that you mock depends on the complexity of the data. If the data is fairly homogenous then imitating a small amount is enough. If the data is heterogenous more mock data to cover various possible cases, that might break the model training pipeline, will be needed.

```
def mock_data(self):
    """Mock data to imitate read from database."""
    mock_data_rdd = self.sc.parallelize([("A", 1, 1), ("B", 1, 0),
                                         ("C", 0, 2), ("D", 2, 4),
                                         ("E", 3, 5)])
    schema = ["id", "x", "y"]
    mock_data_df = self.spark.createDataFrame(mock_data_rdd, schema)
    return mock_data_df
```

We then write a simple test case to check if the count of the mock data is equal to the number of elements created. This will ensure that spark application is running fine and the data is created appropriately. Testing a PySpark application is not easy. The error returned from a failing test is a mixture of Java and Python tracebacks. Hence debugging can get intense. Printing the results or using a debugger like `ipdb` can come in handy.

```
def test_count(self):
    """Check if mock data has five rows."""
    self.assertEqual(len(self.mock_df.collect()), 5)
```

Next we start writing our first actual test. We created a DataFrame (http://spark.apache.org/docs/latest/sql-programming-guide.html) with tuples containing three elements. However, the implementation of k-means (http://spark.apache.org/docs/latest/api/python/_modules/pyspark/ml/clustering.html#KMeans) in MLlib/ML requires a specific data type (http://spark.apache.org/docs/latest/mllib-data-types.html) (DenseVector or SparseVector) as input. Therefore in order to use it in MLlib, we need to convert the data type first. An appropriate test can be to ensure that the data type is expected to be a DenseVector.

Here is the test:

```
def test_convert_df(self):
    """Check if dataframe has the form (id, DenseVector)."""
    input_df = clustering.convert_df(self.spark, self.mock_df)
    self.assertEqual(input_df.dtypes, [("id", "string"),
                                       ("features", "vector")])
```

We can run test-suite with `nosetests` or `pytest`. The test should fail! Now we can go ahead and implement the `convert_df` function, which would make this test pass:

```
def convert_df(spark, data):
    """Transform dataframe into the format that can be used by Spark ML
."""
    input_data = data.rdd.map(lambda x: (x[0], DenseVector(x[1:])))
    df = spark.createDataFrame(input_data, ["id", "features"])
    return df
```

This implementation looks good and we can run the test-suite. Everything should be green! The next step can be to check if we rescale the variables correctly.

```
def test_rescale_df_first_entry(self):
    """Check if rescaling works for the first entry of the first row.""
"
    input_df = clustering.convert_df(self.spark, self.mock_df)
    scaled_df = clustering.rescale_df(input_df)
    self.assertAlmostEqual(scaled_df.rdd.map(lambda x: x.features_scale
d)
                           .take(1)[0].toArray()[0], 0.8770580193070292
)

def test_rescale_df_second_entry(self):
    """Check if rescaling works for the second entry of the first row."
""
    input_df = clustering.convert_df(self.spark, self.mock_df)
    scaled_df = clustering.rescale_df(input_df)
    self.assertAlmostEqual(scaled_df.rdd.map(lambda x: x.features_scale
d)
                           .take(1)[0].toArray()[1], 0.4822428221704121
4)
```

We implement the method, `rescale_df`:

```
def rescale_df(data):
    """Rescale the data."""
    standardScaler = StandardScaler(inputCol="features", outputCol="fea
tures_scaled")
    scaler = standardScaler.fit(data)
    scaled_df = scaler.transform(data)
    return scaled_df
```

The final test could be to check the results of the algorithm. In this case, it is quite easy as our data was artificially made up. In real projects, it might not be simple to formulate the expected results.

```
def test_assign_cluster(self):
    """Check if rows are labeled as expected."""
    input_df = clustering.convert_df(self.spark, self.mock_df)
    scaled_df = clustering.rescale_df(input_df)
    label_df = clustering.assign_cluster(scaled_df)
    self.assertEqual(label_df.rdd.map(lambda x: x.label).collect(),
                     [0, 0, 0, 1, 1])
```

Finally, we can implement `assign_cluster` to make the test pass:

```
def assign_cluster(data):
    """Train kmeans on rescaled data and then label the rescaled data."
""
    kmeans = KMeans(k=2, seed=1, featuresCol="features_scaled", predict
ionCol="label")
    model = kmeans.fit(data)
    label_df = model.transform(data)
    return label_df
```

The implementation above can be further refactored. We will leave the actual implementation to you, though. You might be wondering that we haven't discussed testing reading from database and storing the model. We have written unit tests to test the functionality of clustering. Based on the data store in use you can write integration tests to see if we read and write to the database appropriately.

Conclusion

We hope that you now have a better understanding of how to use TDD in a data science project. Always remember to test each phase of the data science pipeline right from data cleaning, feature extraction, model building, model evaluation to model storing, each individually and as a whole.

If you want to know more:

- Data Science and Test Driven Development (https://www.linkedin.com/pulse/data-science-test-driven-development-sam-savage)
- Test Driven Development in Big Data Pipelines (http://www.illation.com/resources/big-data-australia/2014/10/11/test-driven-development-in-big-data-pipelines)
- Four Ways Data Science Goes Wrong and How Test-Driven Data Analysis Can Help (http://www.predictiveanalyticsworld.com/patimes/four-ways-data-science-goes-wrong-and-how-test-driven-data-analysis-can-help/6947/)
- Thoughtful Machine Learning (https://www.safaribooksonline.com/library/view/thoughtful-machine-learning/9781449374075/)
- Pivotal Labs Guideline on TDD (http://pivotal-guides.cfapps.io/craftsmanship/tdd/)