Bonus Chapter

In the earlier chapters our focus was specific to a particular topic in machine learning. This chapter we have stitched together some more interesting topics that may be useful in day to day operations. We included a briefing of all other topics and concepts that we have not covered.

Hypothesis testing:

Hypothesis is claim or assertion that can be tested. On basis of hypothesis testing we can either reject or leave unchallenged a particular statement. This may be more intuitive with an example. Example of claim can be population who eat breakfast are less prone to diabetes than population who don’t eat breakfast. Another example can churn rate of population who received an offer is less than those who don’t receive the offer. The idea is to check if these assertions are true for population. There are two types of hypothesis.

* Null Hypothesis:

This is represented by H0. It is based on conventional wisdom or experience

* Alternate Hypothesis:

This is represented by Ha. This is often the hypothesis of interest that we are trying to validate

If we have to formulate the hypothesis for above example. We can write hypothesis as following.

H0 : Eating breakfast has no effect on diabetic rate

Ha: Eating breakfast has an effect on diabetic rate

The p-value is used as an alternative to rejection points to provide the smallest level of significance at which the null hypothesis would be rejected. A smaller p-value means that there is stronger evidence in favor of the alternative hypothesis.

Table 10-1: Types of errors

|  |  |  |
| --- | --- | --- |
| Decision\Actual | H0 is True | H0 is False |
| Fail to reject null | Correct | Type II error |
| Reject null | Type I error | Correct |

Probability of Type I error is denoted by α and Probability of Type II error is denoted by β. Power of statistical test is given by 1-β. Cost of Type I error far outweighs cost of Type I error. In most cases we will try to reduce α as much as possible. Consider example of flipping a coin. Effect size can be defined as the difference between heads and tails for n flips. Sample size on the other hand is the number of heads and tails. Effect size and sample size both influence the p-value. This is the reason why we can’t reduce both Type I and Type II errors at once. The way we setup the hypothesis testing is first writing the null and alternate hypothesis. Then we define desired level of significance α (usually 5%) for most tests. Run the test and make a decision based on significance level. If p value is less than significance level we reject the null hypothesis. If p value is more than or equal to significance level we fail to reject the null hypothesis. Pyspark statistics package provides methods to run Pearson’s chi-squared tests.

**Chi-squared test:**

Chi-Square goodness of fit test is a non-parametric test that is used to find out how the observed value of a given occurrence is significantly different from the expected value.  The term goodness of fit is used to compare the observed sample distribution with the expected probability distribution.  Chi-Square goodness of fit test determines how well theoretical distribution (such as normal, binomial, or Poisson) fits the empirical distribution.

O = Observed Value

E = Expected Value

Below is the sample code for verifying the pvalues in a labeled dataset based on chi-squared tests.

*# Import Sparksession*

from pyspark.sql import SparkSession

spark=SparkSession.builder.appName("HypothesisTesting").getOrCreate()

from pyspark.ml.linalg import Vectors

from pyspark.ml.stat import ChiSquareTest

data = [(0.0, Vectors.dense(0.5, 10.0)),

(0.0, Vectors.dense(1.5, 20.0)),

(1.0, Vectors.dense(1.5, 30.0)),

(0.0, Vectors.dense(3.5, 30.0)),

(0.0, Vectors.dense(3.5, 40.0)),

(1.0, Vectors.dense(3.5, 40.0))]

df = spark.createDataFrame(data, ["label", "features"])

r = ChiSquareTest.test(df, "features", "label").head()

print("pValues: " + str(r.pValues))

print("degreesOfFreedom: " + str(r.degreesOfFreedom))

print("statistics: " + str(r.statistics))

Figure 10-1: p-values of test

A screenshot of a cell phone

Description automatically generated

**Kolmogorov-Smirnov test:**

Pyspark also has package that supports Kolmogorov-Smirnov test in basic statistics. First what is KS test. It is test of goodness of fit to a probability density function (PDF). It is nonparametric. It is same the chisquare test for goodness of fit but there is no need to divide the data into arbitrary bins. It tests whether the sampled population has the same probability density functions as another known or theoretical probability density functions. This is 1 sample test

It can be used to test for equality of probability density functions of two sampled populations. This is 2-sided sample test. Hypothesis for such occurance can be written as follows.

H0 : PDF1= PDF2

H1 : PDF1 not equal to PDF2

from pyspark.mllib.stat import Statistics

parallelData = *spark.sparkContext*.parallelize([0.1, 0.15, 0.2, 0.3, 0.25])

# run a KS test for the sample versus a standard normal distribution

testResult = Statistics.kolmogorovSmirnovTest(parallelData, "norm", 0, 1)

*# summary of the test including the p-value, test statistic, and null hypothesis*

*# if our p-value indicates significance, we can reject the null hypothesis*

*# Note that the Scala functionality of calling Statistics.kolmogorovSmirnovTest with*

*# a lambda to calculate the CDF is not made available in the Python API*

print(testResult)

**Output:**

Kolmogorov-Smirnov test summary:

degrees of freedom = 0

statistic = 0.539827837277029

pValue = 0.06821463111921133

Low presumption against null hypothesis: Sample follows theoretical distribution.

Random Data Generation:

Often times we would like to create data to test scenarios and algorithms. PySpark has a handy inbuilt function that can just do it calling simple functions as shown below

*from pyspark.mllib.random import RandomRDDs*

*# Generate a random double RDD that contains 1 million i.i.d. values drawn from the*

*# standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.*

*u = RandomRDDs.normalRDD(spark.sparkContext, 1000000, 10)*

*# Apply a transform to get a random double RDD following `N(1, 4)`.*

*v = u.map(lambda x: 1.0 + 2.0 \* x)*

The above example generates a random double RDD, whose values follows the standard normal distribution N(0, 1), and then map it to N(1, 4)

Sampling:

Sampling is one of the important concepts in machine learning. If the sample dataset is not representative often times the experiments with biased measures carry little to no value. In this section we will introduce types of sampling available in Pyspark.

Simple random sampling (SRS):

In Simple random sampling every row is randomly picked and each row has an equal chance to be picked. sample() aids in performing SRS. We have couple of options within SRS. For demonstration we will use the churn modeling dataset we introduced in chapter 8.

* SRS with replacement

*# Read data*

file\_location = "churn\_modelling.csv"

file\_type = "csv"

infer\_schema = "false"

first\_row\_is\_header = "true"

delimiter = ","

df = spark.read.format(file\_type)\

.option("inferSchema", infer\_schema)\

.option("header", first\_row\_is\_header)\

.option("sep", delimiter)\

.load(file\_location)

*# Simple Random Sampling without replacement*

df=df.select('Geography','NumOfProducts','Age','Gender','Tenure','Exited')

df\_srs\_without\_rep = df.sample(False, 0.5, 23)

df\_srs\_without\_rep.show(10,False)

* SRS without replacement

*# Simple Random Sampling with replacement*

df\_srs\_without\_rep = df.sample(True, 0.5, 23)

df\_srs\_without\_rep.show(10,False)

Stratified sampling:

Every member of the population is grouped into homogeneous subgroups called strata and representative of each group (strata) is chosen.

*# Stratified Sampling*

from pyspark.sql.types import IntegerType

df = df.withColumn("Exited", df["Exited"].cast(IntegerType()))

df.select('Exited').describe().show()

stratified\_sampled = df.sampleBy("Exited", fractions={0: 0.5, 1: 0.5}, seed=23)

stratified\_sampled.select('Exited').describe().show()

Figure 10-2: raw data and Stratified sample distributions respectively

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Difference between coalesce and repartition:

Both coalesce and repartition functions are used to control the number of partitions while saving the data. Here is a quick tip if you are just archiving a small and to medium size data and are in a time crunch use coalesce over repartition. Else always try using the repartition. Here is the reason why:

coalesce uses existing partitions to minimize the amount of data that’s shuffled. This results in partitions of different sizes of data.

Repartition does a full shuffle and creates new partition. This ensures equal size partitions of data. It is also recommended you repartition your data after running filtering queries to improve efficiency of your queries. If you are not repartitioning your data after filtering you may end up with many memory partitions resulting in a poor performance. You also have to keep in mind the cost of full shuffle when repartitioning data. Directed Acyclic Graph DAG scheduler gives us a nice insight into how each operation is applied on data.

Switching between Python and Pyspark:

Pyspark is best suited for large datasets whereas Pandas is faster for smaller datasets. You can use pandas data frames in coherence with Pyspark dataframes in multiple situations to optimize the execution times of your code. We saw a classic example in chapter 8, where we build automated pipelines. For calculating metrics, we used pyspark to do the heavy lifting and once we have manageable data (rows in 10 to 100’s with few columns) we moved them to pandas to efficiently process and calculate metrics.

This is great strategy when you know you will be working with rolled data cubes. You can process the large datasets in pyspark and convert the grouped pyspark data frames into pandas dataframes. Since pandas is more mature than pyspark it will open up a myriad of packages which can be useful in various situations.

Curious character of nulls:

In pyspark there are situations when you are trying to fill a null in column with data and use fillna function. Your fillna works fine but you still end up with nulls in a columns. Yes, you guessed it right, when the datatype of the column you are trying to fill in doesn’t match with fillna value you will still end up with a null value. For example you calculate mean for a integer column which may end up in a float value. Now if you are trying to fill this float value, you may still observe nulls. The workaround is to change the datatype of the column.

Common function Conflicts:

There will be instances where functions such as max exist in both python and pyspark sql functions. There will be an import conflict error if you use a wildcard import from pyspark.sql.functions

Join conditions:

Joins are one of key operations for spark program. In general spark has two cluster communication strategies. Node to Node Strategy involves spark data shuffles across the clusters. Per node Strategy performs broadcast joins. There are 3 main types of joins spark uses

* Sort Merge Join
* Broadcast Join
* Shuffle Hash Join

Sort Merge Join:

It contains couple of steps. First step is to sort the datasets and second operation is to merge the sorted data in the partition by iterating over the elements by join key. This is usually the default setting in spark. To achieve an optimized sort merge, join data frame should be uniformly distributed on the joining columns. If all the join keys are co located, it will join will be further optimized as there is no need for shuffle operations. Sort merge join is preferred when both sides are large

Broadcast Join:

These joins are the most efficient ones in spark. But they are useful for joining a small dataset to huge dataset. In this join smaller table will be broadcasted to all worker nodes. Always make sure that you are broadcasting the smaller table.

Shuffle Hash Join:

This join is based on concept of map reduce. This process maps through the data frames and use the values of the join column as output key. Spark uses a custom function to make sure Shuffle Hash join is better suitable for the given dataset than broadcast join. Creating hash table is a expensive opertaion.

It is always a good practice to verify if there are any nulls that exist in the join key. This mistake can be expensive. You may get a unrelated error such as error occurred while calling o206.ShowString or o64.cacheTable. Another reason can be duplicates. If nulls and duplicates are checked before joining tables you can save some time not running into any of the above mentioned errors.

Use sort merge join when both sides are large and a broadcast join when one side is small. There are 2 major strategies spark used for joins

User defined functions (UDFs):

You may encounter issues with UDF even after successfully importing it in the running file. Again you can encounter a strange ShowString error. The real reason is your UDF is not available to some other worker. You can overcome this error by adding into below line to your script. It enables other workers to reference to your module file.

sc.addPyFile(path\_to\_your\_module.py)

It is also recommended that UDF usage is minimal. This because of deserialization to reserialization operation that it triggers in the backend.

Handle the skewness:

Imbalanced datasets can cause spikes in execution duration. We can observe this in Spark UI backend from the min and max time values. Using evenly distributed datasets can significantly improve the performance

In a SQL join operation, the join key is changed to redistribute data in an even manner so that processing for a partition does not take more time. This technique is called salting.  
Add Column to each side with random int between 0 and `spark.sql.shuffle.partitions`-1 to both sides and then add join clause to include join on generated column above

Using Cache:

Cache is a great tool in the arsenal to speed up the performance. If we use cache for all the objects it can end up in slightly slower storage. When we are using cache, it takes away some of the processing capability of your session. In the storage tab you can observe the size in memory and size on disk distribution.

Persist/ Unpersist:

We know spark uses lazy evaluation for executing commands. Persist is one of the attractive options available in spark store. It gives the ability to store intermediate RDD around the cluster for faster access. For illustration say I have initial dataset that I read into spark and apply a filtering operation. Since spark works on the concept of lazy evaluation filtered data doesn’t contain the executed data. It just maintains the order of DAG that needs to be executed for transformation. If the filtered dataset is going to be consumed by different objects to generate different results. For every single iteration the transformation occurs on filtered df. Imagine there are 100 operations on this filtered df. This can quickly add up the processing time and stretch the operation from few mins to hours for a sizeable data.

Persisting data here can save a lot of computational time. When the filtered df is calculated for the first iteration it is persisted in the memory. In the subsequent runs this data frame in the memory is used for all the operations downstream. After your required downstream operations are completed it is good practice to unpersist the stored data from memory and disk.

Shuffle Partitions:

These partitions exist when shuffling data for join or aggregations. Whenever shuffling huge chunks of data get moved between the partitions. This shuffle can happen within the executor or across the executors. These operations can get expensive very quickly. Imagine we have a small dataframe with 8 initial partitions. By doing a simple group by operation we shoot up the shuffle partition count up to 200. It is because of the default shuffle partition for dataframe is set to 200. You have an opportunity to reset the partitions using the below configurations.

sparkSession**.**conf**.**set**(**"spark.sql.shuffle.partitions"**,**50**)**

If you are not setting these configurations you may underutilize your spark resources.

Use optimal formats:

Spark supports many formats, such as CSV, JSON, XML, parquet, orc, and Avro. The best format for performance is parquet with snappy compression, which is the default in Spark 2.x. Parquet stores data in columnar format and is highly optimized in Spark.

Data serialization

Spark jobs are distributed, so appropriate data serialization is important for the best performance. There are two serialization options for Spark:

* Java serialization is the default.
* Kryo serialization is a newer format and can result in faster and more compact serialization than Java. Kryo requires that you register the classes in your program, and it doesn't yet support all Serializable types.

**What we accomplished so far?**

* We learned the overview of hypothesis testing and different sampling techniques
* We learnt some of the common tips and tricks that can save time in day to day activities
* We also had overview of some performance fine tuning techniques.

Fantastic Job! You have completed your jour journey of this book. We hope you thoroughly enjoyed this ride. As they say there is no end to learning. We anticipate that you take this knowledge forward and create efficient scalable solutions for challenging questions you come across. We look forward to hearing your feedback and thoughts for improving the content of the book. Happy PySparking!!