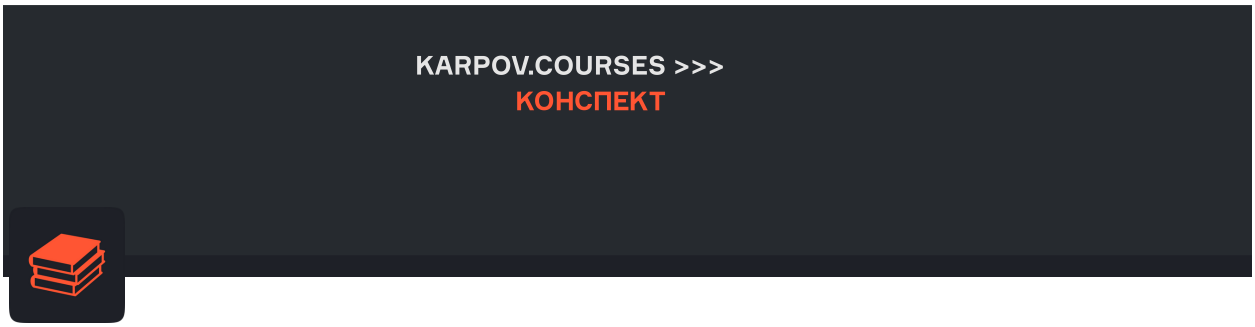


> Конспект > 4 урок > Практика SparkML

> Оглавление

> Подготовка данных

В данной практике мы рассмотрим как с использовать SparkML. Для наглядности будем применять Jupyter.

Сначала импортируем нужные пакеты.

```
from pyspark.sql import SparkSession
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

Создадим SparkSession. Здесь мы задаем имя сессии - `PySparkTitanikJob`.

```
spark = SparkSession.builder.appName("PySparkTitanikJob").getOrCreate()
```

В данной практике мы будем решать задачу классификации пассажиров титаника. Предсказывать мы будем, кто из них погибнет, а кто выживет.

Прочитаем датасет и посмотрим на представленный набор признаков. В данном случае имеющиеся признаки уже почищены от невалидных значений.

```
titanic_df = spark.read.parquet('train.parquet')
titanic_df.show()

+--------+------+------+----+-----+-----+-------+--------+-----------+-----+
|Survived|Pclass|   Sex| Age|SibSp|Parch|   Fare|Embarked|Family_Size|Alone|
+--------+------+------+----+-----+-----+-------+--------+-----------+-----+
|       0|     3|  male|22.0|    1|    0|   7.25|       S|          1|    0|
|       1|     1|female|38.0|    1|    0|71.2833|       C|          1|    0|
|       1|     3|female|26.0|    0|    0|  7.925|       S|          0|    1|
```

Признак `Survived` является целевым, он определяет выживет пассажир или нет. 0 - пассажир погиб, 1 - выжил.

Чтобы модель могла обрабатывать имеющиеся данные нам нужно их подготовить, а именно избавиться от категориальных признаков.

Для этого мы заэнкодим признак `Sex` и `Embarked`. Используем `StringIndexer`, в нем мы задаем входную колонку с категориальны признаком и название новой выходной колонки, где будут уже лежать числовые значения.

```
sex_index = StringIndexer(inputCol='Sex', outputCol="Sex_index")
embarked_index = StringIndexer(
    inputCol='Embarked', outputCol="Embarked_index"
)
```

Чтобы выполнить энкодинг нам нужно вызвать метод `fit` (так как `StringIndexer` является эстиматором), а затем `transform`.

```
titanic_df = sex_index.fit(titanic_df).transform(titanic_df)
titanic_df = embarked_index.fit(titanic_df).transform(titanic_df)
```

Видим, что новые колонки `Sex_index` и `Embarked_index` были успешно созданы.

```
titanic_df.show()
+--------+------+------+...+-----+---------+-------------+
|Survived|Pclass|   Sex|...|Alone|Sex_index|Embarked_index|
+--------+------+------+...+-----+---------+-------------+
|       0|     3|  male|...|    0|      0.0|          0.0|
|       1|     1|female|...|    0|      1.0|          1.0|
|       1|     3|female|...|    1|      1.0|          0.0|
|       1|     1|female|...|    0|      1.0|          0.0|
|       0|     3|  male|...|    1|      0.0|          0.0|
```

Выполним векторизацию для использования эмбэдингов в моделях.

Сначала сформируем список признаков, которые будем использовать. Добавим в него все признаки нашего датасета за исключением целевой колонки `Survived` и колонок с категориальными признаками (`Sex` и `Embarked`).

```
features = ['Pclass', 'Age', 'SibSp', 'SibSp', 'Parch',
            'Fare', 'Alone', 'Sex_index', 'Embarked_index']
```

Создадим трансформер `VectorAssembler` и назначим имя новой колонки `features`, куда будут записаны эмбэдинги. Применим метод `transform`

```
feature = VectorAssembler(inputCols=features, outputCol="features")
feature_vector= feature.transform(titanic_df)
```

Видим, что появилась колонка с фичами.

```
feature_vector.show()
+--------+------+------+...+---------+-------------+--------------------+
|Survived|Pclass|   Sex|...|Sex_index|Embarked_index|            features|
+--------+------+------+---+---------+-------------+--------------------+
|       0|     3|  male|...|      0.0|          0.0|[3.0,22.0,1.0,1.0...|
|       1|     1|female|...|      1.0|          1.0|[1.0,38.0,1.0,1.0...|
|       1|     3|female|...|      1.0|          0.0|[3.0,26.0,0.0,0.0...|
|       1|     1|female|...|      1.0|          0.0|[1.0,35.0,1.0,1.0...|
|       0|     3|  male|...|      0.0|          0.0|(9,[0,1,5,6],[3.0...|
+--------+------+------+...+-----+---------+-------------+-------------+
```

И финальным шагом в подготовке данных для обучения будет создание тренировочной и тестовой выборки. Для этого применим функцию `randomSplit`, указав, что в тренировочный датасет пойдет 80% данных, а в тестовый - 20%, зададим параметр случайности `seed = 42`

Видим, что полученный тренировочная выборка имеет такую же структуру, как и изначальный датасет.

```
(training_data, test_data) = feature_vector.randomSplit([0.8, 0.2],seed = 42)
training_data.show()
+--------+------+------+----+..+---------+-------------+--------------------+
|Survived|Pclass|   Sex| Age|..|Sex_index|Embarked_index|            features|
+--------+------+------+----+..|---------+-------------+--------------------+
|       0|     1|female| 2.0...|      1.0|           0.0|[1.0,2.0,1.0,1.0,...|
|       0|     1|female|25.0|..|      1.0|           0.0|[1.0,25.0,1.0,1.0...|
|       0|     1|  male|18.0|..|      0.0|           1.0|[1.0,18.0,1.0,1.0...|
|       0|     1|  male|19.0|..|      0.0|           0.0|[1.0,19.0,1.0,1.0...|
|       0|     1|  male|19.0|..|      0.0|           0.0|[1.0,19.0,3.0,3.0...|
+--------+------+------+----+---+---------+-------------+--------------------
```

> LogisticRegression

Перейдем к построению и оцениванию модели.

Для этого зададим оценщик `MulticlassClassificationEvaluator`, в него мы подаем:

- `labelCol='Survived'` - колонку, которую мы будем использоваться для сравнения;
- `predictionCol='prediction'` - колонку, где будет лежать предсказание;
- `metricName='accuracy'` - метрику, которая будет использована для оценки качества модели.

```
evaluator = MulticlassClassificationEvaluator(
    labelCol="Survived", predictionCol="prediction", metricName="accuracy"
)
```

Будем обучать модель логистической регрессии. Эта модель нам подходит, так как мы предсказываем признак, который имеет всего два возможных значения.

```
from pyspark.ml.classification import LogisticRegression
# создаем модель и указываем целевую колонку и колонку с эмбэддингами
lr = LogisticRegression(labelCol="Survived", featuresCol="features")

# обучаем модель на тренировочных данных
lrModel = lr.fit(training_data)     # lr - эстиматор, lrModel - трансформер
# применим модель на тестовых данных, получим предсказания
lr_prediction = lrModel.transform(test_data)
lr_prediction.select("prediction", "Survived", "features").show(5)
+----------+--------+--------------------+
|prediction|Survived|            features|
+----------+--------+--------------------+
|       1.0|       0|[1.0,50.0,0.0,0.0...|
|       1.0|       0|(9,[0,1,4,5],[1.0...|
|       1.0|       0|[1.0,24.0,0.0,0.0...|
|       0.0|       0|(9,[0,1,5,6],[1.0...|
|       0.0|       0|(9,[0,1,5,6],[1.0...|
+----------+--------+--------------------+
```

В полученных результатах видим результат предсказания в колонке `prediction`. В первых трех строках видим, что модель ошиблась.

Посмотрим точность модели, используя созданный ранее оценщик.

```
lr_accuracy = evaluator.evaluate(lr_prediction)
print("LogisticRegression [Accuracy] = %g"% (lr_accuracy))
print("LogisticRegression [Error] = %g " % (1.0 - lr_accuracy))

LogisticRegression [Accuracy] = 0.813793
LogisticRegression [Error] = 0.186207
```

Получаем, что:

точность - LogisticRegression [Accuracy] = 0.813793

ошибка - LogisticRegression [Error] = 0.186207

> DecisionTreeClassifier, RandomForestClassifier, Gradient-boosted tree classifier

Применим другую модель `DecisionTreeClassifier`. Код будет выглядет аналогично разобранному выше для логистической регрессии.

```
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(labelCol="Survived", featuresCol="features")
dt_model = dt.fit(training_data)
dt_prediction = dt_model.transform(test_data)

dt_prediction.select("prediction", "Survived", "features").show(5)
+----------+--------+--------------------+
|prediction|Survived|            features|
+----------+--------+--------------------+
|       1.0|       0|[1.0,50.0,0.0,0.0...|
|       0.0|       0|(9,[0,1,4,5],[1.0...|
|       0.0|       0|[1.0,24.0,0.0,0.0...|
|       0.0|       0|(9,[0,1,5,6],[1.0...|
|       0.0|       0|(9,[0,1,5,6],[1.0...|
+----------+--------+--------------------+
```

Видим, что получили уже другие предсказания и с первого взгляда кажется, что они лучше тех, что были получены для логистической регрессии. Но чтобы удостовериться в этом нам нужно использовать оценщик.

```
lr_accuracy = evaluator.evaluate(lr_prediction)
print("LogisticRegression [Accuracy] = %g"% (lr_accuracy))
print("LogisticRegression [Error] = %g " % (1.0 - lr_accuracy))
```

Получаем, что:

точность - DecisionTreeClassifier [Accuracy] = 0.82069

ошибка - DecisionTreeClassifier [Error] = 0.17931

Аналогично можно применять и другие модели, ничего дополнительного для этого не требуется.

Так, например, используем `RandomForestClassifier`:

```
from pyspark.ml.classification import RandomForestClassifier
rf = RandomForestClassifier(labelCol="Survived", featuresCol="features")
rf_model = rf.fit(training_data)
rf_prediction = rf_model.transform(test_data)
rf_prediction.select("prediction", "Survived", "features").show(5)
+----------+--------+--------------------+
```

```
|prediction|Survived|            features|
+----------+--------+--------------------+
|       1.0|       0|[1.0,50.0,0.0,0.0...|
|       0.0|       0|(9,[0,1,4,5],[1.0...|
|       0.0|       0|[1.0,24.0,0.0,0.0...|
|       0.0|       0|(9,[0,1,5,6],[1.0...|
|       0.0|       0|(9,[0,1,5,6],[1.0...|
+----------+--------+--------------------+

rf_accuracy = evaluator.evaluate(rf_prediction)
print("RandomForestClassifier [Accuracy] = %g"% (rf_accuracy))
print("RandomForestClassifier [Error] = %g" % (1.0 - rf_accuracy))
```

Получаем, что:

точность - RandomForestClassifier [Accuracy] = 0.827586

ошибка - RandomForestClassifier [Error] = 0.172414

Попробуем `Gradient-boosted tree classifier`:

```
from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(labelCol="Survived", featuresCol="features",maxIter=10)
gbt_model = gbt.fit(training_data)
gbt_prediction = gbt_model.transform(test_data)
gbt_prediction.select("prediction", "Survived", "features").show(5)
+----------+--------+--------------------+
|prediction|Survived|            features|
+----------+--------+--------------------+
|       1.0|       0|[1.0,50.0,0.0,0.0...|
|       0.0|       0|(9,[0,1,4,5],[1.0...|
|       0.0|       0|[1.0,24.0,0.0,0.0...|
|       1.0|       0|(9,[0,1,5,6],[1.0...|
|       1.0|       0|(9,[0,1,5,6],[1.0...|
+----------+--------+--------------------+
gbt_accuracy = evaluator.evaluate(gbt_prediction)
print("Gradient-boosted [Accuracy] = %g"% (gbt_accuracy))
print("Gradient-boosted [Error] = %g"% (1.0 - gbt_accuracy))
```

Получаем, что

точность - Gradient-boosted [Accuracy] = 0.841379

ошибка - Gradient-boosted [Error] = 0.158621

Видим, что у градиентного бустинга получилась самая высокая точность предсказания.

> Save & Load Model

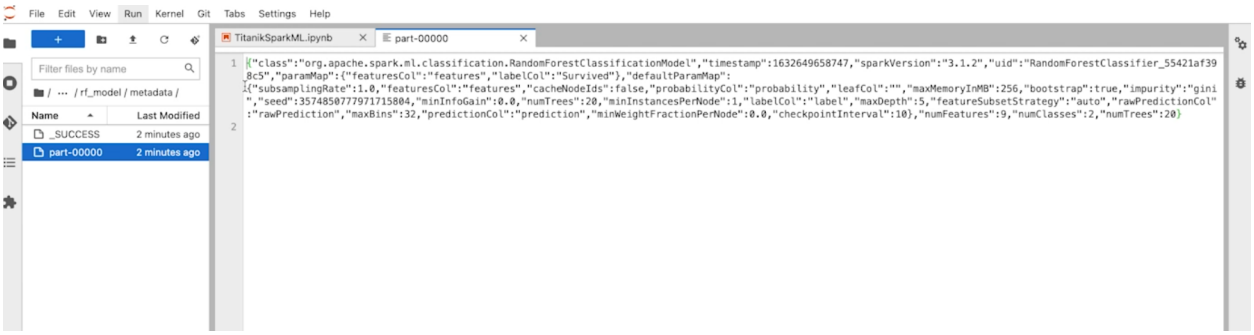
Обученные модели можно сохранять. Для этого надо использовать метод `save`. `overwrite` вызывается для перезаписи устаревшей версии модели с таким же названием.

```
rf_model.write().overwrite().save('rf_model')
```

Сохраненную модель можно загрузить с помощью метода `load`. Для этого нужно использовать именно тот класс модели, который использовался при ее создании.

```
from pyspark.ml.classification import RandomForestClassificationModel
type(RandomForestClassificationModel.load('rf_model'))
```

У сохраненных моделей мы можем прочитать метаданные с атрибутами модели:



> Pipeline

Для сборки модели со всеми ее трансформациями будем использовать Pipeline.

```
from pyspark.ml.pipeline import PipelineModel
# Заново загрузим датафрейм.
# Это нужно для того, чтобы в пайплайн добавить преобразование данных.
titanic_df = spark.read.parquet('train.parquet')
# разделим датасет на тестовую и тренировочную выборку
train, test = titanic_df.randomSplit([0.8, 0.2])

# создадим StringIndexer для категориальных колонок
indexer_sex = StringIndexer(inputCol="Sex", outputCol="Sex_index")
indexer_embarked = StringIndexer(
```

```
    inputCol="Embarked", outputCol="Embarked_index"
)
# создадим колонку с эмбедингами
feature = VectorAssembler(
    inputCols=["Pclass", "Age", "SibSp", "Parch", "Fare",
               "Family_Size","Embarked_index","Sex_index"],
    outputCol="features"
)
# но сразу применять трансформацию данных пока не будем

# выберем модель, которую будем обучать
rf_classifier = RandomForestClassifier(
    labelCol="Survived", featuresCol="features"
)
```

Трансформации мы описали, теперь создадим сам пайплайн.

```
pipeline = Pipeline(
    stages=[indexer_sex, indexer_embarked, feature, rf_classifier]
)
```

Стейджы будут применятся в указанной последовательности,
т.е `rf_classifier` будет выполнен последним.

Обучим модель.

```
p_model = pipeline.fit(train)
```

Если посмотреть на тип `p_model`, то мы увидим, что
это `pyspark.ml.pipeline.PipelineModel`, эту модель можно использовать так же как и любые другие модели.

Сохраним модель и загрузим.

```
p_model.write().overwrite().save('p_model')
model = PipelineModel.load('p_model')
```

Полученная модель является трасформером, поэтому для ее применения надо вызвать метод `transform`.

Заметим, что тестовый датасет не содержит всех нужных для модели колонок, потому что они будут добавлены только при применении модели к датафрейму.

```
prediction = p_model.transform(test)
test.show(5)
+--------+------+----+----+-----+-----+--------+--------+-----------+-----+
|Survived|Pclass| Sex| Age|SibSp|Parch|    Fare|Embarked|Family_Size|Alone|
+--------+------+----+----+-----+-----+--------+--------+-----------+-----+
|       0|     1|male|24.0|    0|    0|    79.2|       C|          0|    1|
|       0|     1|male|24.0|    0|    1|247.5208|       C|          1|    0|
|       0|     1|male|28.0|    0|    0|    47.1|       S|          0|    1|
|       0|     1|male|28.0|    1|    0| 82.1708|       C|          1|    0|
|       0|     1|male|30.0|    0|    0|     0.0|       S|          0|    1|
+--------+------+----+----+-----+-----+--------+--------+-----------+-----+
```

Посмотрим на предсказания. Видим, что в результирующей таблице имеются столбцы, которые были добавлены при применении модели.

```
prediction.select([
    "Pclass", "Age", "SibSp", "Parch", "Fare", "Family_Size",
    "Embarked_index","Sex_index","prediction"
]).show(5)
+------+---+-----+-----+-----+-----------+--------------+---------+----------+
|Pclass|Age|SibSp|Parch| Fare|Family_Size|Embarked_index|Sex_index|prediction|
+------+---+-----+-----+-----+-----------+--------------+---------+----------+
|     1| 24|    0|    0| 79.2|          0|           1.0|      0.0|       0.0|
|     1| 24|    0|    1|247.5|          1|           1.0|      0.0|       0.0|
|     1| 28|    0|    0| 47.1|          0|           0.0|      0.0|       0.0|
|     1| 28|    1|    0|82.17|          1|           1.0|      0.0|       0.0|
|     1| 30|    0|    0|  0.0|          0|           0.0|      0.0|       0.0|
+------+---+-----+-----+-----+-----------+--------------+---------+----------+

prediction.printSchema()

root
 |-- Survived: integer (nullable = true)
 |-- Pclass: integer (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Age: double (nullable = true)
 |-- SibSp: integer (nullable = true)
 |-- Parch: integer (nullable = true)
 |-- Fare: double (nullable = true)
 |-- Embarked: string (nullable = true)
 |-- Family_Size: integer (nullable = true)
 |-- Alone: integer (nullable = true)
 |-- Sex_index: double (nullable = false)
 |-- Embarked_index: double (nullable = false)
 |-- features: vector (nullable = true)
```

```
 |-- rawPrediction: vector (nullable = true)
 |-- probability: vector (nullable = true)
 |-- prediction: double (nullable = false)
```

Оценим модель:

```
evaluator = MulticlassClassificationEvaluator(
    labelCol="Survived", predictionCol="prediction", metricName="accuracy"
)
p_accuracy = evaluator.evaluate(prediction)
print("Pipeline model [Accuracy] = %g"% (p_accuracy))
print("Pipeline model [Error] = %g " % (1.0 - p_accuracy))
```

Получили, что

точность - Pipeline model [Accuracy] = 0.834254

ошибка - Pipeline model [Error] = 0.165746

> Hyperparameter tuning

Для подбора гиперпараметров зададим сетку. Зададим возможные значения для параметров `maxDepth`, `maxBins`, `minInfoGain`.

```
from pyspark.ml.tuning import ParamGridBuilder, TrainValidationSplit
paramGrid = ParamGridBuilder() \
    .addGrid(rf_classifier.maxDepth, [2, 3, 4]) \
    .addGrid(rf_classifier.maxBins, [4, 5, 6]) \
    .addGrid(rf_classifier.minInfoGain, [0.05, 0.1, 0.15]) \
    .build()
```

Объявим `TrainValidationSplit`, указывая эстиматор (наш пайплайн), `estimatorParamMaps` - сетка параметров, `evaluator` - оценщик, `trainRatio` - доля разбивки на тестовую и обучающую выборку.

```
tvs = TrainValidationSplit(estimator=pipeline,
                           estimatorParamMaps=paramGrid,
                           evaluator=evaluator,
                           trainRatio=0.8)
```

Запусим процесс обучения модели, вызвав метод `fit`

```
model = tvs.fit(train)
```

Тип такой модели будет `pyspark.ml.tuning.TrainValidationSplitModel`

Такая модель по умолчанию будет применять ту обученную версию, которая показала лучшие результаты.

Тип такой лучшей модели (`type(model.bestModel)`) будет уже обычный `pyspark.ml.pipeline.PipelineModel`

Посмотрим на гиперпараметры, которые получились в процессе оптимизации:

```
# обращаемся к последнему стейджу, так как там и лежит обученная модель
jo = model.bestModel.stages[-1]._java_obj
print('Max Depth: {}'.format(jo.getMaxDepth()))
print('Num Trees: {}'.format(jo.getMaxBins()))
print('Impurity: {}'.format(jo.getMinInfoGain()))
```

- `Max Depth:` 2
- `Trees`: 4
- `Impurity`: 0.1