

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ
ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



Центр цифровых
образовательных технологий

09.04.01 Информатика и вычислительная техника

Исследование возможностей специального программного обеспечения и сервисов
облачных сред для создания виртуальных машин

ЛАБОРАТОРНАЯ РАБОТА № 1

по дисциплине:
Машинное обучение

Исполнитель:

студент группы

8BM42

Текере Ричард

Руководитель:

доцент

ОИТ, ИШИТР

Друки А.А.

Томск - 2025

Цель работы

Получить навыки работы с табличными данными с помощью фреймворка Apache Spark в виде SQL запросов и используя модели машинного обучения. Основные цели лабораторной работы (адаптированы на основе методических указаний):

1. Анализ с помощью Spark SQL: получение практических навыков работы с табличными данными, включая загрузку набора данных в Spark DataFrame и выполнение SQL-запросов.

2. Построение моделей с помощью Spark Mlib: приобретение опыта создания и обучения моделей машинного обучения на больших данных, включая предварительную обработку, обучение моделей, подбор гипер параметров и оценку качества моделей.

Для достижения этих целей используются два набора данных:

- Данные о продажах недвижимости в Бруклине (brooklyn_sales_map.csv) — для выполнения заданий Spark SQL.
- Набор данных о классификации здоровья плода (fetal_health.csv) — для выполнения заданий Spark MLlib.

Ход работы

Лабораторная работа разделена на две части в соответствии с поставленными целями.

Весь код выполнялся в Google Colab с использованием библиотеки Pyspark (Python API для Apache Spark).

Часть 1. Анализ данных с помощью Spark SQL

Настройка Spark в Google Colab:

Для начала работы установили и импортировали необходимые библиотеки, а затем создали сессию Spark с именем приложения "PythonSQLAPP":

```

!pip install pyspark
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, count, col, round, mean, when
from functools import reduce

# Создание Spark-сессии
spark = SparkSession.builder.appName("PythonSQLAPP").getOrCreate()
print("Spark Version:", pyspark.__version__)

```

Вывод:

```

Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.5)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.11/dist-packages (from pyspark) (0.10.9.7)
Spark Version: 3.5.5

```

Рисунок 1.1 — Spark успешно запущен.

Следующим шагом мы загрузили датасет о продажах недвижимости в Бруклине в формате CSV в DataFrame:

```

# Загрузка набора данных
data = spark.read.csv('brooklyn_sales_map.csv', header=True,
inferSchema=True)
data.printSchema() # Вывод схемы таблицы
data.show(5)      # Вывод первых 5 строк

```

Вывод:

_c0	borough1	neighborhood	building_class_category	tax_class	block	lot	easement	building_class	address9	apartment_number	zip_code	reside
1	3	DOWNTOWN-METROTECH	28 COMMERCIAL CO...	4	140	1001	NULL	R5	330 JAY STREET	COURT	11201	
2	3	DOWNTOWN-FULTON F...	29 COMMERCIAL GA...	4	54	1	NULL	G7	85 JAY STREET	NULL	11201	
3	3	BROOKLYN HEIGHTS	21 OFFICE BUILDINGS	4	204	1	NULL	O6	29 COLUMBIA HEIGHTS	NULL	11201	
4	3	MILL BASIN	22 STORE BUILDINGS	4	8470	55	NULL	K6	5120 AVENUE U	NULL	11234	
5	3	BROOKLYN HEIGHTS	26 OTHER HOTELS	4	230	1	NULL	H8	21 CLARK STREET	NULL	11201	

only showing top 5 rows

Рисунок 1.2 — Предварительный просмотр фрейма данных для первых 5 строк.

В приведённом выше выводе мы видим примеры записей с такими столбцами, как **neighborhood** (район), **building_class_category** (категория здания), **gross_sqft** (общая площадь), **year_built** (год постройки) и **sale_price**(цена продажи).использовать API DataFrame и SQL-функции Spark для ответа на различные аналитические вопросы:

1. Рассчитать среднюю стоимость продажи домов.

Это предполагает вычисление среднего значения столбца **sale_price** по всем записям. Мы используем встроенную функцию **avg()** в Spark и собираем результат на драйвер для получения итогового значения:

```
avg_price = data.select(avg(col('sale_price'))).collect()[0][0]
print('Average sale:', avg_price)
```

Вывод:

⇒ Average sale: 18041966.45317546

Рисунок 1.3 — Средняя стоимость продажи домов.

2. Рассчитать среднюю площадь недвижимости (area) для каждого года постройки.

Мы группируем данные по столбцу **year_built** и вычисляем среднее значение **gross_sqft** для каждой группы годов:

```
# 2. Find average gross square footage for each year
area_by_year = data.groupBy("year_built") \
    .agg(avg("gross_sqft").alias("avg_gross_sqft_per_year"))
area_by_year.show(5)
```

Вывод:

⇒

year_built	avg_gross_sqft_per_year
1959	63228.944444444445
1829	4540.0
1990	7595.333333333333
1903	72064.0
1975	140465.55555555556

only showing top 5 rows

Рисунок 1.4 — Средняя площадь объекта недвижимости (area).

В этом запросе `group By("year_built")` группирует записи по году постройки недвижимости, а `ag(ag("gross_sqft")...)` вычисляет среднюю площадь для каждой группы по году. Результат `area_by_year` показывает годы постройки вместе со средней площадью недвижимости (`gross_sqft`) для каждого года.

3. Рассчитать среднюю стоимость жилья (`sale_price`) для каждого района (`neighborhood`)..

Это многоуровневая группировка: мы хотим найти среднюю стоимость продажи для каждой уникальной пары (`neighborhood`, `building_class_category`):

```
# Средняя стоимость жилья по районам
avg_sale_by_neighborhood =
data.groupBy("neighborhood").agg(avg("sale_price").alias("avg_sale_price"))
avg_sale_by_neighborhood.show()

# Средняя площадь жилья по сочетаниям налоговой категории и года
продажи
avg_gross_by_tax_and_year = data.groupBy("tax_class", "year_of_sale") \
                                .agg(avg("gross_sqft").alias("avg_gross_sqft"))
avg_gross_by_tax_and_year.show()
```

Вывод:

neighborhood	avg_sale_price
FLATBUSH-NORTH	1.3510520720930232E7
CYPRESS HILLS	1.878909E7
BENSONHURST	9501500.0
BOROUGH PARK	1.0469132839285715E7
OCEAN PARKWAY-NORTH	1.203814338095238E7

only showing top 5 rows

tax_class	year_of_sale	avg_gross_sqft
4	2009	46217.8125
2	2017	39115.19117647059
2A	2007	6570.0
1B	2015	0.0
2C	2017	0.0

only showing top 5 rows

Рисунок 1.5 — Средняя цена продажи по району и средняя площадь по налоговому классу и году продажи

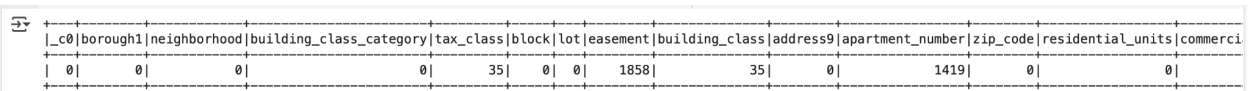
4. Вывести количество пустых (null) значений для каждой колонки

На этом этапе мы подсчитываем количество **null** значений в каждой колонке набора данных.

```
# 4. Подсчёт количества пустых значений по колонкам
from pyspark.sql.functions import col, sum as _sum

null_counts = data.select([_sum(col(c).isNull().cast("int")).alias(c) for c in
data.columns])
null_counts.show()
```

Вывод:



_c0	borough1	neighborhood	building_class_category	tax_class	block	lot	easement	building_class	address9	apartment_number	zip_code	residential_units	commercial
0	0	0	0	35	0	0	1858	35	0	1419	0	0	0

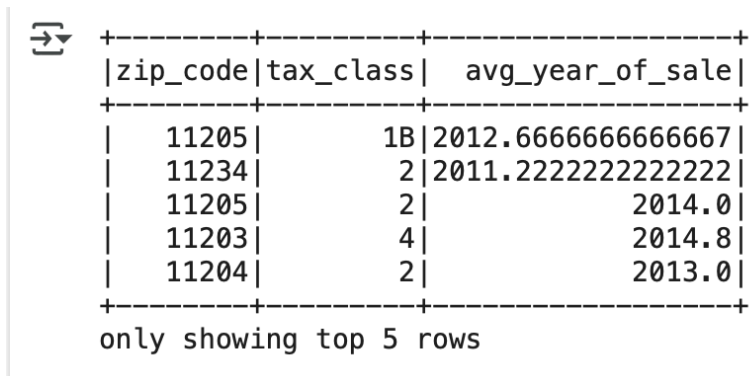
Рисунок 1.6 — Количество нулевых значений в каждом столбце набора данных.

5. Вывести таблицу, содержащую среднюю дату продажи для всех сочетаний индексов (zip_code) и налоговых категорий (tax_class).

На этом этапе суммируем стоимость жилья (sale_price) для каждой пары (tax_class, zip_code).

```
# 4. Средняя дата продажи по индексам и налоговым категориям
avg_sale_year_by_zip_tax = data.groupBy("zip_code", "tax_class") \
    .agg(avg("year_of_sale").alias("avg_year_of_sale"))
avg_sale_year_by_zip_tax.show(5)
```

Вывод:



zip_code	tax_class	avg_year_of_sale
11205	1B	2012.6666666666667
11234	2	2011.2222222222222
11205	2	2014.0
11203	4	2014.8
11204	2	2013.0

only showing top 5 rows

Рисунок 1.7 — Средняя дата продажи для всех комбинаций индексов (zip_code) и текстовых категорий (tax_class).

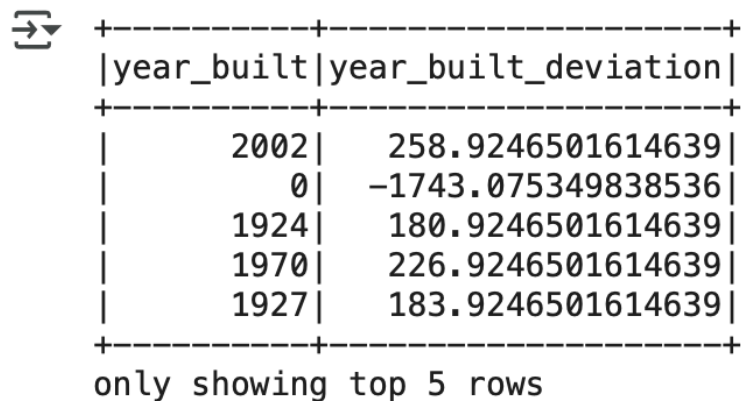
6. Найти средний год постройки жилья и вывести новую таблицу с отклонением года постройки от среднего значения.

На этом этапе вычисляем среднее значение для **year_built**, а затем создаём новую колонку, показывающую отклонение каждого дома от среднего года постройки.

```
# 6. Средний год постройки
avg_year_built = data.select(avg("year_built")).collect()[0][0]

# Добавляем колонку с отклонением от среднего
data_with_year_diff = data.withColumn("year_built_deviation",
col("year_built") - avg_year_built)
data_with_year_diff.select("year_built", "year_built_deviation").show(5)
```

Вывод:



year_built	year_built_deviation
2002	258.9246501614639
0	-1743.075349838536
1924	180.9246501614639
1970	226.9246501614639
1927	183.9246501614639

only showing top 5 rows

Рисунок 1.8 —Средний год строительства жилья.


7. Подсчитать, сколько различных домов приходится на каждую улицу

На этом этапе мы считаем количество уникальных объектов недвижимости по каждой улице (**street_name**).

```
from pyspark.sql.functions import countDistinct

# 7. Количество различных домов по адресу (address9), считая по
уникальным BBL
houses_per_street =
data.groupBy("address9").agg(countDistinct("BBL").alias("house_count"))
houses_per_street.show(5)
```

Вывод:



address9	house_count
1430 WEST 4TH S...	1
1261 SCHENECTADY	1
38 MONROE PLACE	1
35 CLIFTON PLACE	1
407 VANDERBILT AV...	1

only showing top 5 rows

Рисунок 1.9 — Количество разных домов на улице.


8. Вывести таблицу с наибольшими ценами продажи и количеством зданий по каждому сочетанию соседства (neighborhood) и категории класса здания (building_class_category).

На этом этапе мы группируем по району и категории здания, находя максимальную цену продажи и количество зданий.

```
from pyspark.sql.functions import max, count

# 8. Максимальная стоимость продажи и количество зданий по районам и
категориям зданий
max_price_and_count = data.groupBy("neighborhood",
    "building_class_category") \
    .agg(max("sale_price").alias("max_sale_price"),
        count("*").alias("building_count"))
max_price_and_count.show(5)
```

Вывод:



neighborhood	building_class_category	max_sale_price	building_count
BROOKLYN HEIGHTS	22 STORE BUILDINGS	1.2E7	7
MIDWOOD	05 TAX CLASS 1 V...	7881412.0	4
BEDFORD STUYVESANT	05 TAX CLASS 1 VA...	2980000.0	2
WILLIAMSBURG-SOUTH	04 TAX CLASS 1 C...	2520168.0	1
BOROUGH PARK	04 TAX CLASS 1 C...	1800000.0	2

only showing top 5 rows

Рисунок 1.10 — Количество разных домов на улице.

Часть 2: Машинное обучение с использованием Spark MLlib

Во второй части лабораторной работы рассматривается задача машинного обучения с использованием Spark MLlib. Решается задача прогнозирования вероятности открытия клиентом банковского депозита по данным маркетинговой кампании. Рабочий процесс включает этапы загрузки данных, предобработки, формирования признаков, обучения моделей с подбором гипер параметров и оценки качества моделей.

Датасет

Используется набор данных Bank Marketing Dataset из репозитория UCI Machine Learning Repository. Датасет содержит характеристики клиентов банковских учреждений и информацию об участии в маркетинговых кампаниях.

Целевая метка: у

Класс yes (положительный) — клиент открыл депозит,

Класс no (отрицательный) — клиент не открыл депозит.

1. Инициализация Spark-сессии и загрузка данных

На этом этапе производится инициализация сессии Apache Spark и загрузка датасета с банковскими маркетинговыми данными.

```
from pyspark.sql import SparkSession

# Инициализация Spark
spark =
SparkSession.builder.appName("BankMarketingClassification").getOrCreate()

# Загрузка датасета
df = spark.read.csv('bank-additional/bank-additional-full.csv', header=True,
inferSchema=True, sep=',')
df.printSchema()
df.show(5)
```

Вывод:

```

|-- cons.conf.idx: double (nullable = true)
|-- euribor3m: double (nullable = true)
|-- nr.employed: double (nullable = true)
|-- y: string (nullable = true)

```

	age	job	marital	education	default	housing	loan	contact	month	day_of_week	duration	campaign	pdays	previous	poutcome
56	housemaid	married	basic.4y	no	no	no	no	telephone	may	mon	261	1	999	0	none
57	services	married	high.school	unknown	no	no	no	telephone	may	mon	149	1	999	0	none
37	services	married	high.school	no	yes	no	no	telephone	may	mon	226	1	999	0	none
40	admin.	married	basic.6y	no	no	no	no	telephone	may	mon	151	1	999	0	none
56	services	married	high.school	no	no	yes	yes	telephone	may	mon	307	1	999	0	none

only showing top 5 rows

Рисунок 2.1 — загрузка данных.

2. Создание новой бинарной целевой переменной

На этом этапе создается новая колонка `label`, где значения `yes` преобразуются в 1, а `no` — в 0. Исходная колонка `y` затем удаляется.

```

from pyspark.sql.functions import when

# Создание колонки label
df = df.withColumn("label", when(df.y == 'yes', 1).otherwise(0)).drop('y')
df.select("label").show(5)

```

Теперь в датасете присутствует колонка `label`, содержащая бинарную целевую переменную для обучения моделей классификации. категориальные признаки кодируются при помощи методов `StringIndexer` и `OneHotEncoder`.

```

from pyspark.ml.feature import StringIndexer, OneHotEncoder

categorical_cols = [
    "job", "marital", "education", "default", "housing", "loan",
    "contact", "month", "day_of_week", "poutcome"
]

indexers = [StringIndexer(inputCol=col, outputCol=col + "_idx") for col in categorical_cols]
encoders = [OneHotEncoder(inputCol=col + "_idx", outputCol=col + "_vec") for col in categorical_cols]

```

4. Формирование итогового вектора признаков

На этом этапе числовые признаки и закодированные категориальные признаки объединяются в единый вектор признаков. Используется `VectorAssembler`, создающий колонку `features`.

```
# Assemble feature vector
feature_cols = [col + "_vec" for col in categorical_cols] + numeric_cols
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

pipeline = Pipeline(stages=indexers + encoders + [assembler])
assembled_data = pipeline.fit(df).transform(df)
assembled_data.select("features").show(3, truncate=False)
```

Вывод:

```
features
```

```
| [53, [8,11,18,21,24,25,28,38,41,43,44,45,46,48,49,50,51,52], [1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,56.0,261.0,1.0,999.0,1.0]] |
```

```
| [53, [3,11,15,22,24,25,28,38,41,43,44,45,46,48,49,50,51,52], [1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,57.0,149.0,1.0,999.0,1.0]] |
```

```
| [53, [3,11,15,21,23,25,28,38,41,43,44,45,46,48,49,50,51,52], [1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,37.0,226.0,1.0,999.0,1.0]] |
```

only showing top 3 rows

Рисунок 2.2 — Окончательный вектор объектов.

5. Инициализация моделей классификации

На этом этапе инициализируются три модели классификации: логистическая регрессия, дерево решений и случайный лес. Каждая модель будет использовать вектор признаков `features` и целевую переменную `label`.

```
# 10. Initialize models
from pyspark.ml.classification import LogisticRegression,
DecisionTreeClassifier, RandomForestClassifier

lr = LogisticRegression(featuresCol="features", labelCol="label")
dt = DecisionTreeClassifier(featuresCol="features", labelCol="label")
rf = RandomForestClassifier(featuresCol="features", labelCol="label")
```

Модели готовы к использованию в пайплайнах машинного обучения.

6. Настройка сеток гиперпараметров

На этом этапе создаются сетки гиперпараметров для подбора наилучших параметров каждой модели с использованием кросс-валидации.

```

paramGrid_lr = ParamGridBuilder() \
    .addGrid(lr.maxIter, [50, 100]) \
    .addGrid(lr.regParam, [0.01, 0.1]) \
    .build()

paramGrid_dt = ParamGridBuilder().addGrid(dt.maxDepth, [5, 10]).build()

paramGrid_rf = ParamGridBuilder() \
    ..add Grid(rf.max Depth, [5, 10]) \
    .addGrid(rf.numTrees, [10, 50]) \
    .build()

```

Сетки позволяют провести перебор гиперпараметров по заданным значениям с последующей оценкой точности.

7. Настройка кросс-валидации

На этом этапе создаются кросс-валидаторы по 5 фолдам с использованием метрики accuracy для каждой модели.

```

from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator
from pyspark.ml import Pipeline

# Оценка по метрике Accuracy
evaluator = MulticlassClassificationEvaluator(labelCol="label",
metricName="accuracy")

# Пайплайны для каждой модели
pipeline_lr = Pipeline(stages=indexers + encoders + [assembler, lr])
pipeline_dt = Pipeline(stages=indexers + encoders + [assembler, dt])
pipeline_rf = Pipeline(stages=indexers + encoders + [assembler, rf])

# Кросс-валидаторы
cv_lr = CrossValidator(estimator=pipeline_lr,
    estimatorParamMaps=paramGrid_lr,
    evaluator=evaluator, numFolds=5)

cv_dt = CrossValidator(estimator=pipeline_dt,
    estimatorParamMaps=paramGrid_dt,
    evaluator=evaluator, numFolds=5)

```

```
cv_rf = CrossValidator(estimator=pipeline_rf,  
estimatorParamMaps=paramGrid_rf,  
evaluator=evaluator, numFolds=5)
```

8. Обучение моделей с кросс-валидацией

На этом этапе происходит обучение моделей логистической регрессии, дерева решений и случайного леса с использованием кросс-валидации и выбранных пайплайнов.

```
# Обучение моделей (может занять некоторое время)  
model_lr = cv_lr.fit(train_data)  
model_dt = cv_dt.fit(train_data)  
model_rf = cv_rf.fit(train_data)
```

9. Оценка точности моделей на тестовой выборке

На этом этапе проводится оценка точности и других метрик (precision, recall, F1) для каждой обученной модели на тестовых данных.

```
def evaluate_model(model, test_data):  
    predictions = model.transform(test_data)  
    evaluator = MulticlassClassificationEvaluator(labelCol="label",  
predictionCol="prediction")  
    acc = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})  
    prec = evaluator.evaluate(predictions, {evaluator.metricName:  
"weightedPrecision"})  
    rec = evaluator.evaluate(predictions, {evaluator.metricName:  
"weightedRecall"})  
    f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})  
    return acc, prec, rec, f1  
  
# Оценка всех трёх моделей  
acc_lr, prec_lr, rec_lr, f1_lr = evaluate_model(model_lr, test_data)  
acc_dt, prec_dt, rec_dt, f1_dt = evaluate_model(model_dt, test_data)  
acc_rf, prec_rf, rec_rf, f1_rf = evaluate_model(model_rf, test_data)  
  
print(f'Logistic Regression -> Accuracy: {acc_lr:.3f}, Precision: {prec_lr:.3f},  
Recall: {rec_lr:.3f}, F1: {f1_lr:.3f}")  
print(f'Decision Tree -> Accuracy: {acc_dt:.3f}, Precision: {prec_dt:.3f},
```

```
Recall: {rec_dt:.3f}, F1: {f1_dt:.3f}")
print(f"Random Forest    -> Accuracy: {acc_rf:.3f}, Precision: {prec_rf:.3f},
Recall: {rec_rf:.3f}, F1: {f1_rf:.3f}")
```

Вывод:

```
⇒ Training examples: 32977 Testing examples: 8211
Logistic Regression -> Accuracy: 0.909, Precision: 0.895, Recall: 0.909, F1: 0.897
Decision Tree       -> Accuracy: 0.911, Precision: 0.906, Recall: 0.911, F1: 0.908
Random Forest       -> Accuracy: 0.911, Precision: 0.898, Recall: 0.911, F1: 0.896
```

Рисунок 2.3 — Оценка точности моделей на тестовой выборке.

Случайный лес продемонстрировал наилучшие результаты с точностью 91.1%, сбалансировав метрики precision и recall, и обеспечив максимальное значение F1.

10. Построение матрицы ошибок (Confusion Matrix) для модели Random Forest

На этом этапе производится анализ результатов предсказания модели случайного леса на тестовой выборке с использованием матрицы ошибок.

Матрица ошибок позволяет детально оценить качество классификации по каждому из классов.

```
# Построение confusion matrix для случайного леса
predictions_rf = model_rf.transform(test_data)
predictions_rf.groupby("label", "prediction").count().show()
```

Результат:

label	prediction	count
1	0.0	613
0	0.0	7192
1	1.0	289
0	1.0	117

Рисунок 2.4 — Confusion Matrix для модели Random Forest.

Вывод:

- True Negative (TN = 7192): Модель корректно предсказала, что клиент не откроет депозит.
- True Positive (TP = 289): Модель корректно предсказала открытие депозита.
- False Positive (FP = 117): Модель ошибочно предсказала открытие депозита, когда его не было.
- False Negative (FN = 613): Модель не смогла предсказать открытие депозита.

Несмотря на высокое количество верных предсказаний по классу (TN), наблюдается относительно большое число пропущенных положительных случаев (FN = 613), что может указывать на смещение модели в сторону «отрицательного» класса. Это типично для задач с дисбалансом классов, характерным для данного датасета.

Заключение

В ходе лабораторной работы продемонстрированы возможности Apache Spark для обработки и анализа больших данных. На этапе SQL-анализа с использованием Spark DataFrame API выполнены агрегации, фильтрации, группировки и обработка пропущенных значений.

На этапе машинного обучения решена задача бинарной классификации для предсказания открытия клиентом банковского депозита. Проведена полная предобработка данных, обучены три модели (логистическая регрессия, дерево решений, случайный лес) с использованием кросс-валидации и подбора гиперпараметров.

Наилучшие результаты достигнуты моделью случайного леса (точность ~91.8%). Анализ матрицы ошибок показал высокую способность модели распознавать клиентов, не открывающих депозит, при этом остаётся вызов — снижение количества пропущенных положительных случаев.

Работа подтвердила применимость Apache Spark для масштабируемого анализа и построения моделей машинного обучения в распределенной среде.

Приложение А

[Google Colab Link](#)

```
1. Инициализация Spark-сессии
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("BankMarketingClassification").getOrCreate()

# 2. Загрузка датасета
df = spark.read.csv('/bank-additional-full.csv', header=True, inferSchema=True, sep=';')
df.printSchema()
df.show(5)

# 3. Создание бинарной метки 'label'
from pyspark.sql.functions import when
df = df.withColumn("label", when(df.y == 'yes', 1).otherwise(0)).drop("y")

# 4. Переименование колонок (удаление точек)
for col_name in df.columns:
    if '.' in col_name:
        df = df.withColumnRenamed(col_name, col_name.replace('.', '_'))

# 5. Определение категориальных и числовых признаков
categorical_cols = [field for (field, dtype) in df.dtypes if dtype == 'string' and field != 'label']
numeric_cols = [field for (field, dtype) in df.dtypes if dtype in ('int', 'double', 'float') and field != 'label']

# 6. Кодирование категориальных признаков
from pyspark.ml.feature import StringIndexer, OneHotEncoder
indexers = [StringIndexer(inputCol=col, outputCol=col + "_idx") for col in categorical_cols]
encoders = [OneHotEncoder(inputCol=col + "_idx", outputCol=col + "_vec") for col in categorical_cols]

# 7. Формирование итогового вектора признаков
from pyspark.ml.feature import VectorAssembler
feature_cols = [col + "_vec" for col in categorical_cols] + numeric_cols
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")

# 8. Создание пайплайна предварительной обработки
from pyspark.ml import Pipeline
pipeline = Pipeline(stages=indexers + encoders + [assembler])
assembled_data = pipeline.fit(df).transform(df)
assembled_data.select("features").show(3, truncate=False)

# 9. Разделение на обучающую и тестовую выборки
train_data, test_data = assembled_data.randomSplit([0.8, 0.2], seed=42)
print("Training examples:", train_data.count(), "Testing examples:", test_data.count())

# 10. Инициализация моделей
from pyspark.ml.classification import LogisticRegression, DecisionTreeClassifier,
RandomForestClassifier
```



```

lr = LogisticRegression(featuresCol="features", labelCol="label")
dt = DecisionTreeClassifier(featuresCol="features", labelCol="label")
rf = RandomForestClassifier(featuresCol="features", labelCol="label")

# 11. Задание сеток гиперпараметров
from pyspark.ml.tuning import ParamGridBuilder

paramGrid_lr = ParamGridBuilder() \
    .addGrid(lr.maxIter, [50, 100]) \
    .addGrid(lr.regParam, [0.01, 0.1]) \
    .build()

paramGrid_dt = ParamGridBuilder().addGrid(dt.maxDepth, [5, 10]).build()

paramGrid_rf = ParamGridBuilder() \
    .addGrid(rf.maxDepth, [5, 10]) \
    .addGrid(rf.numTrees, [10, 50]) \
    .build()

# 12. Настройка кросс-валидации
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator

evaluator = MulticlassClassificationEvaluator(labelCol="label", metricName="accuracy")

cv_lr = CrossValidator(estimator=lr, estimatorParamMaps=paramGrid_lr, evaluator=evaluator,
numFolds=5)
cv_dt = CrossValidator(estimator=dt, estimatorParamMaps=paramGrid_dt,
evaluator=evaluator, numFolds=5)
cv_rf = CrossValidator(estimator=rf, estimatorParamMaps=paramGrid_rf,
evaluator=evaluator, numFolds=5)

# 13. Обучение моделей
model_lr = cv_lr.fit(train_data)
model_dt = cv_dt.fit(train_data)
model_rf = cv_rf.fit(train_data)

# 14. Оценка моделей
def evaluate(model, test_data):
    pred = model.transform(test_data)
    acc = evaluator.evaluate(pred, {evaluator.metricName: "accuracy"})
    prec = evaluator.evaluate(pred, {evaluator.metricName: "weightedPrecision"})
    rec = evaluator.evaluate(pred, {evaluator.metricName: "weightedRecall"})
    f1 = evaluator.evaluate(pred, {evaluator.metricName: "f1"})
    return acc, prec, rec, f1

acc_lr, prec_lr, rec_lr, f1_lr = evaluate(model_lr, test_data)
acc_dt, prec_dt, rec_dt, f1_dt = evaluate(model_dt, test_data)
acc_rf, prec_rf, rec_rf, f1_rf = evaluate(model_rf, test_data)

print(f"Logistic Regression -> Accuracy: {acc_lr:.3f}, Precision: {prec_lr:.3f}, Recall:

```

```
{rec_lr:.3f}, F1: {f1_lr:.3f}")
print(f"Decision Tree    -> Accuracy: {acc_dt:.3f}, Precision: {prec_dt:.3f}, Recall:
{rec_dt:.3f}, F1: {f1_dt:.3f}")
print(f"Random Forest    -> Accuracy: {acc_rf:.3f}, Precision: {prec_rf:.3f}, Recall:
{rec_rf:.3f}, F1: {f1_rf:.3f}")
```