phase2a

January 11, 2023

1 TBD - etap 2a

1.0.1 Michał Kopyt, Rafał Kulus, Adrian Prorok

1.1 Wczytanie danych

[1]:
```
pip install googledrivedownloader
```

```
Requirement already satisfied: googledrivedownloader in
/opt/conda/lib/python3.10/site-packages (0.4)
Note: you may need to restart the kernel to use updated packages.
```

[2]:
```
from google_drive_downloader import GoogleDriveDownloader as gdd
from pathlib import Path

path_dir = str(Path.home()) + "/data/2020/"  # ustawmy sciezke na HOME/data/2020
archive_dir = path_dir + "survey.zip"        # plik zapiszemy pod nazwa survey.
 ↪zip

# sciagniecie pliku we wskazane miejsce
gdd.download_file_from_google_drive(file_id='1dfGerWeWkcyQ9GX9x20rdSGj7WtEpzBB',
                                    dest_path=archive_dir,
                                    unzip=True)
```

1.2 Podłączenie do sesji Spark

[3]:
```
from pyspark.sql import SparkSession

spark = SparkSession \
.builder \
.config("spark.executor.instances", "1")\
.config('spark.driver.memory','1g')\
.config('spark.executor.memory', '1g') \
.getOrCreate()
```

1.3 Dostęp do danych

[4]:
```
import os
user_name = 'jovyan'
# ścieżka dostępu do pliku
csv_path = 'file:////home/jovyan/data/2020/survey_results_public.csv'
df = spark.read.csv(csv_path, inferSchema=True, header="true", nullValue='NA',␣
 ↪nanValue='NA',emptyValue='NA')
```

1.4 a) Ile jest unikalnych odpowiedzi w zapytaniu o poziom wykształcenia (EdLevel)?

[5]:
```
df_ed_level = df.select(df.EdLevel).distinct()

print(f'Liczba unikalnych odpowiedzi w zapytaniu o poziom wyksztalcenia:␣
 ↪{df_ed_level.count()}')
```

```
Liczba unikalnych odpowiedzi w zapytaniu o poziom wyksztalcenia: 10
```

1.5 b) Podaj średnią liczbę godzin przepracowywanych przez respondentów pogrupowanych ze względu na kraj.

[6]:
```
df.groupBy("Country").avg("WorkWeekHrs").show()
```

```
+--------------------+------------------+
|             Country|  avg(WorkWeekHrs)|
+--------------------+------------------+
|       Côte d'Ivoire|              32.0|
|            Paraguay|            29.875|
|The former Yugosl…|              39.5|
|               Yemen|              40.0|
|             Senegal|34.285714285714285|
|              Sweden|  40.7496062992126|
|  Hong Kong (S.A.R.)|42.298507462686565|
|   Republic of Korea|35.138888888888886|
|         Philippines| 37.32920792079208|
|             Eritrea|              null|
|           Singapore| 40.25684931506849|
|            Malaysia|39.255639097744364|
|              Turkey| 43.77202643171806|
|              Malawi|              36.5|
|                Iraq| 49.07142857142857|
|             Germany| 40.27298744460857|
|         Afghanistan|              46.5|
|            Cambodia|              44.0|
|              Jordan|45.806451612903224|
|            Maldives|              47.3|
```

```
+--------------------+-----------------+
only showing top 20 rows
```

1.6 c) Narysuj wykres słupkowy popularności wykorzystywanych baz danych przez profesjonalnych programistów. Skorzystaj z funkcji split i posexplode.

[7]:
```
df_databases = df.select("DatabaseWorkedWith").filter("DatabaseWorkedWith is␣
 ↪not NULL")
df_databases.show()
```

```
+--------------------+
|  DatabaseWorkedWith|
+--------------------+
|Elasticsearch;Mic…|
|MySQL;PostgreSQL;…|
| MariaDB;MySQL;Redis|
|Microsoft SQL Server|
|Firebase;MongoDB;…|
|Firebase;Microsof…|
|        MySQL;Oracle|
|          PostgreSQL|
|Microsoft SQL Ser…|
|Elasticsearch;Mar…|
|MariaDB;Microsoft…|
|IBM DB2;MariaDB;M…|
|Firebase;MariaDB;…|
|   PostgreSQL;SQLite|
|              Oracle|
|          PostgreSQL|
|Elasticsearch;MyS…|
|Microsoft SQL Ser…|
|        MySQL;SQLite|
|MongoDB;MySQL;Pos…|
+--------------------+
only showing top 20 rows
```

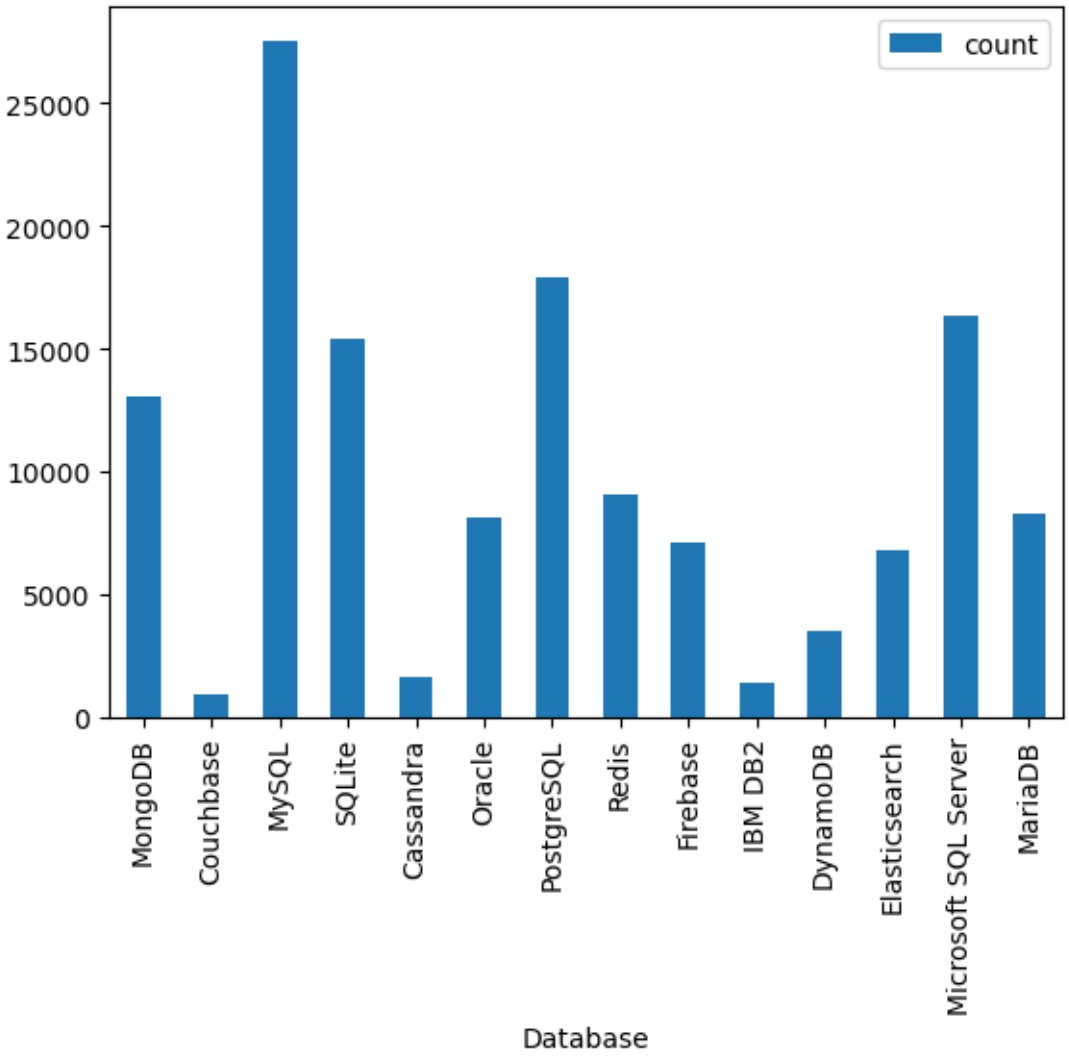
[8]:
```
from pyspark.sql.functions import split, col, posexplode

df_databases_splitted = df_databases.
 ↪select(posexplode(split(col("DatabaseWorkedWith"), ";")).alias("Position",␣
 ↪"Database"))
df_databases_splitted.show()
```

```
+--------+--------------------+
|Position|            Database|
+--------+--------------------+
```

```
|       0|       Elasticsearch|
|       1|Microsoft SQL Server|
|       2|              Oracle|
|       0|               MySQL|
|       1|          PostgreSQL|
|       2|               Redis|
|       3|              SQLite|
|       0|             MariaDB|
|       1|               MySQL|
|       2|               Redis|
|       0|Microsoft SQL Server|
|       0|            Firebase|
|       1|             MongoDB|
|       2|          PostgreSQL|
|       3|              SQLite|
|       0|            Firebase|
|       1|Microsoft SQL Server|
|       0|               MySQL|
|       1|              Oracle|
|       0|          PostgreSQL|
+--------+--------------------+
only showing top 20 rows
```

[9]:
```
df_databases_splitted.groupBy("Database").count().toPandas().plot.
  ↪bar(x="Database")
```

[9]: <AxesSubplot: xlabel='Database'>



1.7 Przygotowanie modelu do predykcji

W ramach zadania chcemy stworzyć klasyfikator, który będzie przewidywać czy respondent zarabia więcej niż 60000 USD rocznie.

[10]:
```
db_name = user_name.replace('-','_')
spark.sql(f'DROP DATABASE IF EXISTS {db_name} CASCADE')
spark.sql(f'CREATE DATABASE {db_name}')
spark.sql(f'USE {db_name}')
table_name = "survey_2020"
spark.sql(f'DROP TABLE IF EXISTS {table_name}')
spark.sql(f'CREATE TABLE IF NOT EXISTS {table_name} \
          USING csv \
          OPTIONS (HEADER true, INFERSCHEMA true, NULLVALUE "NA") \
```

```
          LOCATION "{csv_path}"')

spark_df= spark.sql(f'SELECT *, CAST((convertedComp > 60000) AS STRING) AS␣
 ↪compAboveAvg \
                    FROM {table_name} where convertedComp IS NOT NULL ')
```

[11]:
```
from pyspark.ml.feature import StringIndexer, OneHotEncoder, VectorAssembler
from pyspark.ml import Pipeline

y = 'compAboveAvg'
feature_columns = ['OpSys', 'EdLevel', 'MainBranch' , 'Country', 'JobSeek',␣
 ↪'YearsCode']

stringindexer_stages = [StringIndexer(inputCol=c, outputCol='stringindexed_' +␣
 ↪c).setHandleInvalid("keep") for c in feature_columns]
stringindexer_stages += [StringIndexer(inputCol=y, outputCol='label')]

onehotencoder_stages = [OneHotEncoder(inputCol='stringindexed_' + c,␣
 ↪outputCol='onehot_' + c) for c in feature_columns]
extracted_columns = ['onehot_' + c for c in feature_columns]
vectorassembler_stage = VectorAssembler(inputCols=extracted_columns,␣
 ↪outputCol='features')

final_columns = [y] + feature_columns + extracted_columns + ['features',␣
 ↪'label']

transformed_df = Pipeline(stages=stringindexer_stages + \
                          onehotencoder_stages + \
                          [vectorassembler_stage]).fit(spark_df).
 ↪transform(spark_df).select(final_columns)

training, test = transformed_df.randomSplit([0.8, 0.2], seed=1234)
```

1.7.1 Drzewo decyzyjne

[12]:
```
# na poczatek wybierzemy drzewo decyzyjne. Nie musimy podawac zadnych parametrow
from pyspark.ml.classification import DecisionTreeClassifier
dt = DecisionTreeClassifier(featuresCol='features', labelCol='label')

simple_model = Pipeline(stages=[dt]).fit(training)

pred_simple = simple_model.transform(test)

# macierz pomyłek (confusion matrix)
label_and_pred = pred_simple.select('label', 'prediction')
label_and_pred.groupBy('label', 'prediction').count().toPandas()
```

[12]:

	label	prediction	count
0	1.0	1.0	2259
1	0.0	1.0	644
2	1.0	0.0	831
3	0.0	0.0	3132

[13]:
```
# Ewaluator
from pyspark.ml.evaluation import BinaryClassificationEvaluator

evaluator = BinaryClassificationEvaluator(rawPredictionCol="rawPrediction",␣
 ↪metricName="areaUnderROC")

auroc_simple = evaluator.evaluate(pred_simple)

from pyspark.ml.evaluation import MulticlassClassificationEvaluator

evaluator_m = MulticlassClassificationEvaluator(labelCol="label",␣
 ↪predictionCol="prediction", metricName="accuracy")
accuracy = evaluator_m.evaluate(pred_simple)

print(f'Drzewo decyzyjne, auroc: {auroc_simple}, accuracy: {accuracy}')
```

```
Drzewo decyzyjne, auroc: 0.5986083542455158, accuracy: 0.7851733177978445
```

1.7.2 Dodanie hiperparametrów

[14]:
```
# Jakie wartości hiperparametru maxDepth mają być przetestowane:
from pyspark.ml.tuning import ParamGridBuilder

param_grid = ParamGridBuilder().\
    addGrid(dt.maxDepth, [2,3,4,5,6]).\
    build()

# Walidacja krzyżowa wykonwyana w celu optymalizacji hiperparametrów
from pyspark.ml.tuning import CrossValidator

cv = CrossValidator(estimator=dt, estimatorParamMaps=param_grid,␣
 ↪evaluator=evaluator, numFolds=4)

# Budowa modelu na podstawie danych treningowych
cv_model = cv.fit(training)

cv_model.bestModel
```

[14]:
```
DecisionTreeClassificationModel: uid=DecisionTreeClassifier_5bf042cf4950,
depth=2, numNodes=5, numClasses=2, numFeatures=229
```

1.7.3 Predykcja z nowym modelem

[15]:
```
pred_cv = cv_model.transform(test)

# Confusion matrix
label_and_pred = pred_cv.select('label', 'prediction')
label_and_pred.groupBy('label', 'prediction').count().toPandas()
```

[15]:

	label	prediction	count
0	1.0	1.0	1816
1	0.0	1.0	345
2	1.0	0.0	1274
3	0.0	0.0	3431

[16]:
```
auroc_cv = evaluator.evaluate(pred_cv)
auroc_cv
```

[16]:
```
0.6893249307498216
```

[17]:
```
acc_cv = evaluator_m.evaluate(pred_cv)
acc_cv
```

[17]:
```
0.7642004078065832
```

1.7.4 Klasyfikacja za pomoca Gradient Boosted Trees

[18]:
```
from pyspark.ml.classification import GBTClassifier
gbt = GBTClassifier(labelCol="label", featuresCol="features", maxIter=10)
model = gbt.fit(training)

auroc_gbt = evaluator.evaluate(model.transform(test))
acc_gbt = evaluator_m.evaluate(model.transform(test))

print(f'GBT, auroc: {auroc_gbt}, accuracy: {acc_gbt}')
```

```
GBT, auroc: 0.88280847183369, accuracy: 0.8049810661229245
```

1.8 d1) Sprawdź czy można bardziej poprawić jakość predykcji dla zadania z tego notatnika: dodając cechy?

Do dodanych wcześniej atrybutów dodaliśmy cechy: NEWOvertime i YearsCodePro

[19]:
```
feature_columns = ['OpSys', 'EdLevel', 'MainBranch' , 'Country', 'JobSeek',␣
 ↪'YearsCode']
feature_columns += ['NEWOvertime', 'YearsCodePro']

stringindexer_stages = [StringIndexer(inputCol=c, outputCol='stringindexed_' +␣
 ↪c).setHandleInvalid("keep") for c in feature_columns]
```

```
stringindexer_stages += [StringIndexer(inputCol=y, outputCol='label')]

onehotencoder_stages = [OneHotEncoder(inputCol='stringindexed_' + c,␣
 ↪outputCol='onehot_' + c) for c in feature_columns]
extracted_columns = ['onehot_' + c for c in feature_columns]
vectorassembler_stage = VectorAssembler(inputCols=extracted_columns,␣
 ↪outputCol='features')
final_columns = [y] + feature_columns + extracted_columns + ['features',␣
 ↪'label']

transformed_df = Pipeline(stages=stringindexer_stages + \
                          onehotencoder_stages + \
                          [vectorassembler_stage]).fit(spark_df).
 ↪transform(spark_df).select(final_columns)

training, test = transformed_df.randomSplit([0.8, 0.2], seed=1234)

# Drzewo decyzyjne
dt = DecisionTreeClassifier(featuresCol='features', labelCol='label')

simple_model = Pipeline(stages=[dt]).fit(training)

pred_simple = simple_model.transform(test)

# macierz pomyłek (confusion matrix)
label_and_pred = pred_simple.select('label', 'prediction')
label_and_pred.groupBy('label', 'prediction').count().toPandas()
```

[19]:

	label	prediction	count
0	1.0	1.0	2254
1	0.0	1.0	608
2	1.0	0.0	836
3	0.0	0.0	3168

[20]:
```
auroc_simple = evaluator.evaluate(pred_simple)
accuracy = evaluator_m.evaluate(pred_simple)

print(f'Drzewo decyzyjne, dodane cechy, auroc: {auroc_simple}, accuracy:␣
 ↪{accuracy}')
```

```
Drzewo decyzyjne, dodane cechy, auroc: 0.6254565540837035, accuracy:
0.7896883192542965
```

[21]:
```
# GBT
model = gbt.fit(training)
pred_gbt = model.transform(test)
auroc_gbt = evaluator.evaluate(pred_gbt)
```

```
accuracy_gbt = evaluator_m.evaluate(pred_gbt)

print(f'GBT dodane cechy, auroc: {auroc_gbt}, accuracy: {accuracy_gbt}')
```

```
GBT dodane cechy, auroc: 0.8959790329658275, accuracy: 0.8204194581998252
```

Zarówno dla drzewa decyzyjnego jak i GBT, dodanie nowych cech pozwoliło nieznacznie poprawić wartość miar auroc i accuracy.

1.9 d2) Sprawdź czy można bardziej poprawić jakość predykcji dla zadania z tego notatnika: zmieniając model?

[22]:
```
from pyspark.ml.classification import LogisticRegression
lr = LogisticRegression(labelCol="label", featuresCol="features")
model_lr = lr.fit(training)
pred_lr = model.transform(test)

auroc_lr = evaluator.evaluate(pred_lr)
accuracy_lr = evaluator_m.evaluate(pred_lr)

print(f'Regresja logistyczna, dodane cechy, auroc: {auroc_lr}, accuracy:␣
 ↪{accuracy_lr}')
```

```
Regresja logistyczna, dodane cechy, auroc: 0.8959790329658275, accuracy:
0.8204194581998252
```

Zmiana modelu na LogisticRegression daje podobne wartości miar auroc i accuracy, co GradientBoostedTree.

1.10 d3) Sprawdź czy można bardziej poprawić jakość predykcji dla zadania z tego notatnika: lepiej dobierając parametry nowego modelu?

[23]:
```
param_grid = ParamGridBuilder() \
    .addGrid(model_lr.regParam,[0.0, 0.1, 0.3, 0.6]) \
    .addGrid(model_lr.elasticNetParam, [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]) \
    .build()

cv_lr = CrossValidator(estimator=lr, estimatorParamMaps=param_grid,␣
 ↪evaluator=evaluator, numFolds=4)

# Budowa modelu na podstawie danych treningowych
cv_model_lr = cv_lr.fit(training)

cv_model.bestModel
```

[23]:
```
DecisionTreeClassificationModel: uid=DecisionTreeClassifier_5bf042cf4950,
depth=2, numNodes=5, numClasses=2, numFeatures=229
```

[24]:
```
pred_lr_best = cv_model.bestModel.transform(test)

auroc_lr = evaluator.evaluate(pred_lr_best)
accuracy_lr_best = evaluator_m.evaluate(pred_lr_best)

print(f'Regresja logistyczna, dostrojone parametry, dodane cechy, auroc:␣
  ↪{auroc_lr}, accuracy: {accuracy_lr}')
```

```
Regresja logistyczna, dostrojone parametry, dodane cechy, auroc:
0.6893249307498216, accuracy: 0.8204194581998252
```

W tym wypadku optymalizacja parametrów nie poprawiła wyników modelu.

[25]:
```
spark.stop()
```