PREDVIĐANJE ZAGAĐENJA I CLUSTERIZACIJA BIG MOBILITY PODATAKA U APACHE SPARKU

Danilo Milošević 1732

PODACI

- Koristimo isti skup podataka kao u prethodnom zadatku, ali samo za izduvne gasove
- 1 CSV fajla veličine ~ 1GB.
- Fajlovi sadrže podatke o izduvnim gasovima kao i geografskoj lokaciji automobila, motora, bicikli i pešaka

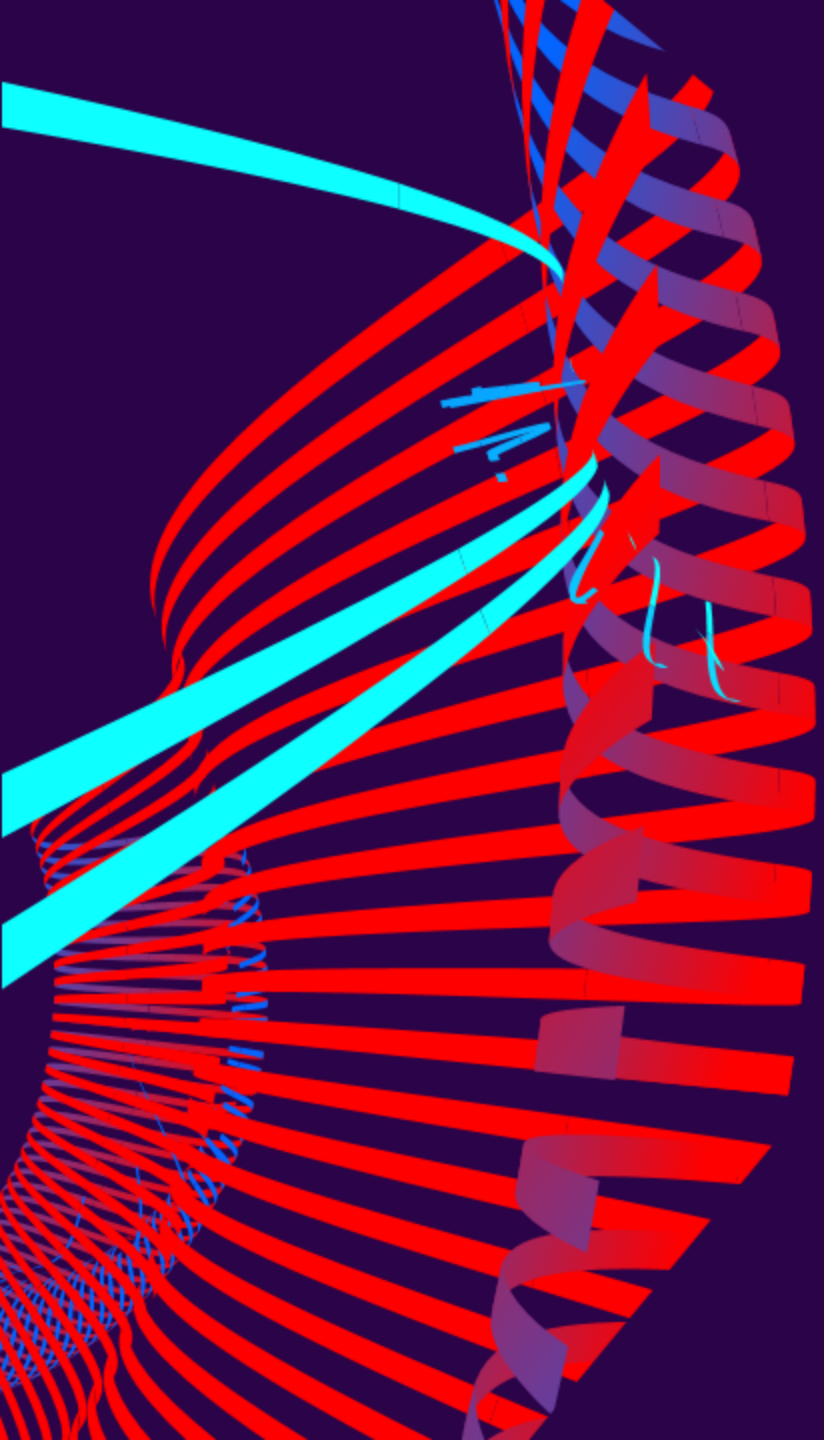
PREGLED DELOVA

Producer

- Python skripta koja šalje podatke na Kafka stream
- Može se izabrati Kafka stream u zavisnosti da li su geografski ili emisioni podaci

Spark Consumer

- Python skripte koja vrši treniranje I predikciju zagađenja I klasterizaciju



PRODUCER

PRODUCER

Konfiguracija

- __init__ funkcijom kreiramo instancu klase Producer gde se poziva i određivanje konfiguracije
- U okviru funkcije get_app_config određujemo gru pu kao i topice na koji se subscribujemo.
- Postoje dva topic-a, jedan sa podacima za trening i jedan za test podatke
- Pored toga se određuju i drugi argumenti kao što su
  - putanja CSV fajla,
  - Koliko podataka ide u train skup
  - da li štampamo poruke na konzolu
  - Error flag, indikuje da li je konfiguracija ispravna

```python
class Producer:
    def __init__(self, args, createProducer = True):
        self.config={}
        self.get_app_config(args=args)
        if not self.config['error'] and createProducer:
            self.producer = self.configure_producer()

    def configure_producer(self):
        producer_config = {
            'bootstrap_servers': '0.0.0.0:9094',
            'client_id': self.config['client_id'],
            'acks': 'all',
            'linger_ms': 10,
        }
        return KafkaProducer(**producer_config)
```

- U okviru get_app_config funkcije se takođe vrši provera unetih argumenata
- Ukoliko argumenti fale konfiguracija je označena da poseduje error
- Ukoliko su argumenti pogrešno upisani korisnik se obaveštava i postavlja se error flag.

```python
def get_app_config(self, args):
    start_path = '/Users/danilomilosevic/Documents/Danilo/VS/'
    self.config = {
        'client_id':'emission-producer',
        'group':'emission-group',
        'to_print':False,
        'sleep_time':0,#s
        'train_topic':'ems_train_topic',
        'pred_topic':'ems_pred_topic',
        'file':start_path + 'emissions.csv',
        'error':False,
        'split_train_pred': 5000
    }

    if len(args) < 2:
        print("\tUsage: python3 producer.py [print|noprint] [sleep_time(ms)]")
        self.config['error'] = True
        return

    try:
        self.config['to_print'] = args[1] == 'print'
        try:
            self.config['sleep_time'] = float(args[5])/1000.0
        except ValueError:
            self.config['error'] = True
            print('\tSleep time has to be in milliseconds!')
    except:
        pass
```

PRODUCER

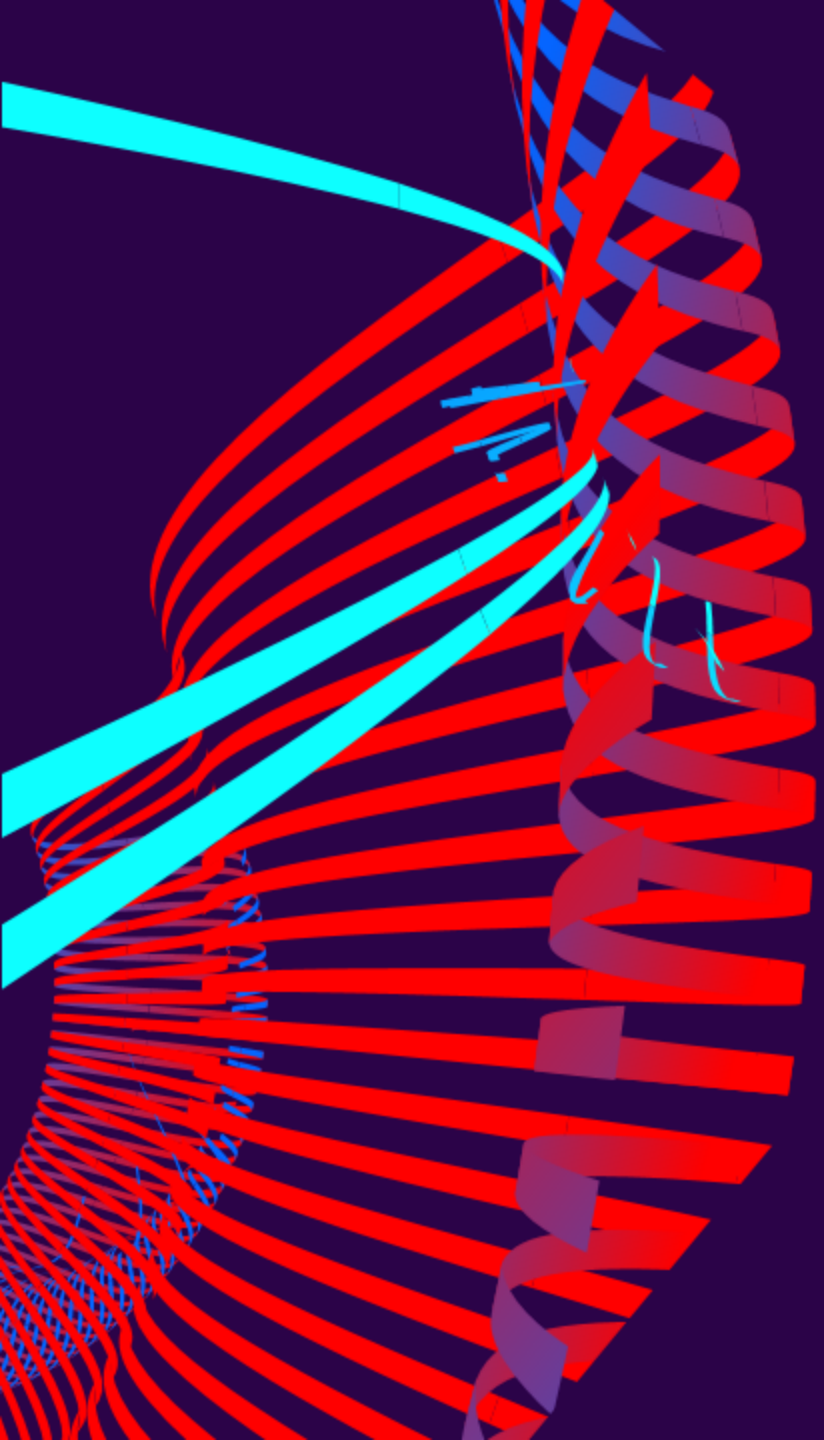
Record production

- U okviru funkcije produce_records se obavlja sledeće:
  - Proverava se greška u konfiguraciji i štampa odgovarajuća poruka
  - Otvara se fajl i preskače prva linija (CSV header)
  - Generišemo random početni datum kao timestamp
  - Čita se linija po liniju
  - Linija se formatira u specifičnom formatu
  - Linija se šalje na topic, pri čemu se odlučuje da li se šalje na train ili predict topic
  - Ukoliko je print flag true, štampamo na ekran šta se šalje

```
def produce_records(self):
    if(self.config['error']):
        print('\tError in config!')
        return
    curr = 0
    with open(self.config['file']) as file:
        line = file.readline() #skip first line
        start_date = Producer.get_random_date()
        line = file.readline()
        while line:
            processed_line, ok = self.format_line(line, start_date=start_date)
            if ok:
                if self.config['to_print']:
                    print('Sending to topic: ', processed_line)
                self.producer.send(
                    topic = self.config['train_topic'] if curr < self.config['split_train_pred'] else self.config['pred_topic'],
                    value = processed_line.strip().encode('utf-8'))
                self.producer.flush()
                if self.config['to_print']:
                    print('\tSent!')
                curr+=1
            time.sleep(self.config['sleep_time'])
            line = file.readline()
    self.producer.flush()
    self.producer.close()
```

- Podaci se formatiraju u funkciji format_line
- Dodaje se timestamp kao datum i vreme

```
def format_line(self, line, start_date=None):
    seconds = float(line.split(";")[0])
    return (Producer.get_date_timestamp(start_date, seconds) + line[line.index(";"):],True)
```



CONSUMER

SPARK CONSUMER

Konfiguracija

- U okviru funkcije get_app_config određujemo grupu kao i topice na koji se subscribujemo.
- Postoje dva topica – jedan sa train i drugi sa predict podacima
- Pored toga se određuju i drugi argumenti kao što su
  - Labela za koju se vrši regresija
  - Niz input feature-a (fuel, electricity, timestamp, noise, speed, x, y, ...)
  - Koliko niti koristi aplikacija
  - Da li postoji greška u konfiguraciji

- Kao i u produceru, pribavljanje argumenata se vrši uz proveru da li su argumenti uneti ispravno
- Ukoliko nisu koristi se default-na vrednost

```python
@staticmethod
def get_app_config(args):
    config = {
        'train_topic':'ems_train_topic',
        'predict_topic':'ems_pred_topic',
        'group':'emission-group',
        'to_print':False,
        'error':False,
        'train_inputs_ems':['timestamp_ms','fuel','electricity','noise','speed'],
        'train_inputs_clusters':['timestamp_ms','x','y'],
        'label':'CO',
        'num_proc':'*'
    }
    i=1
    try:
        if args[i] == '--label':
            i+=1
            config['label'] = str(args[i])
            i+=1

        if (args[i]=="--print"):
            config['to_print'] = True
            i+=1

        if(args[i]=='--n'):
            i+=1
            try:
                config['num_proc'] = str(int(args[i]))
            except:
                config['num_proc'] = '*'
            i+=1
    except Exception as e:
        pass
    return config
```

SPARK CONSUMER

Consume

- U okviru funkcije consume se obavlja sledeće:
  - Uspostavlja se stream sa train topica na kafka:9092
  - Uspostavlja se stream sa predict topica na kafka:9092
  - Kreiramo dva Predictora – Clustering i Regression predictore
  - Podaci za trening i prediction se parse-uju. Trening podaci se filteruju tako da količina izduvnih gasova bude validna vrednost
  - Nakon toga se podaci preprocesiraju odnosno cast-uju u odgovarajuće tipive
  - Podaci se prikupljaju za trening
  - Zatim kreće treniranje modela kao i čuvanje istreniranih modela
  - Na kraju koristimo modele za predikciju i rezultate čuvamo u CSV
  - Kada se program prekine štampa se vreme izvršenja

```python
def consume(self, config):
    start = time.time()
    #Create spark app and datastream
    try:
        spark = SparkSession.builder.appName("EMS").master(f"local[{config['num_proc']}]").getOrCreate() # type: ignore
        kafkaDF = spark.readStream.format("kafka")\
            .option("kafka.bootstrap.servers", "kafka:9092")\
            .option("subscribe", config['train_topic'])\
            .option("startingOffsets", "earliest")\
            .load()
        dataStream_train = kafkaDF.selectExpr("CAST(value AS STRING)").alias("value")
        kafkaDF_pred = spark.readStream.format("kafka")\
            .option("kafka.bootstrap.servers", "kafka:9092")\
            .option("subscribe", config['predict_topic'])\
            .option("startingOffsets", "earliest")\
            .load()
        dataStream_pred = kafkaDF_pred.selectExpr("CAST(value AS STRING)").alias("value")
    except Exception as e:
        print("Failed to create data stream: ", e)
        exit(1)

    #Create a predictor
    try:
        predict_ems: Predictor = RegressionPredictor('label','features')
        predict_cluster: Predictor = ClusteringPredictor(5)
    except Exception as e:
        print("Failed to create a predictor: ",e)
        exit(1)

    #Parse the data
    try:
        train_data = ConsumerSpark.select_emissions(dataStream_train)
        train_data = train_data.filter(col(config['label']) > 0)

        pred_data = ConsumerSpark.select_emissions(dataStream_pred)
    except Exception as e:
        print("Failed to parse data: ",e)
        exit(1)
```

```python
#Preprocess the data
try:
    train_ems = ConsumerSpark.process_emissions(train_data)
    pred_ems = ConsumerSpark.process_emissions(pred_data)

    train_cluster = ConsumerSpark.process_region(train_data)
    pred_cluster = ConsumerSpark.process_region(pred_data)
except Exception as e:
    print("Failed to process data: ", e)
    exit(1)

train_ems = train_ems.writeStream.format("memory").queryName("train_ems").start().awaitTermination(10)
train_ems = spark.sql("select * from train_ems")

train_cluster = train_cluster.writeStream.format("memory").queryName("train_cluster").start().awaitTermination(10)
train_cluster = spark.sql("select * from train_cluster")

#Train
print("Train...")
try:
    self.train_model(predict_ems, train_ems, config['train_inputs_ems'], config['label'])
    self.train_model(predict_cluster, train_cluster, config['train_inputs_clusters'], config['label'])
except Exception as e:
    print("Failed to train the model: ", e)

print("Predict...")
try:
    predicted_ems = self.predict(predict_ems, pred_ems, config['train_inputs_ems'], config['label'])
    predicted_cong = self.predict(predict_cluster, pred_cluster, config['train_inputs_clusters'], '')

    query_ems = predicted_ems \
            .writeStream \
            .outputMode("append") \
            .format("csv") \
            .option("path", f"./{config['label']}_reggresion") \
            .option("checkpointLocation", "./checkpoint_regression") \
            .start()
    query_cong = predicted_cong \
            .writeStream \
            .outputMode("append") \
            .format("csv") \
            .option("path", f"./clusterization") \
            .option("checkpointLocation", "./checkpoint_clusters") \
            .start()
    query_ems.awaitTermination()
    query_cong.awaitTermination()
except Exception as e:
    end = time.time()
    print("Time in s: ",(end-start))
    print("Failed to predict using the model: ", e)
```

SPARK CONSUMER

Formatiranje

- U funkciji select_emissions biramo kolone koje koristimo tokom treninga
- Funkcijama process_region i process_emissions castujemo vrednosti i biramo one koje su nam bitne za trening

```
@staticmethod
def select_emissions(stream:SparkSession): # type: ignore
    return stream.selectExpr("split(value, ';') as parsed").selectExpr( # type: ignore
            "parsed[0] AS timestamp",
            "parsed[1] AS CO",
            "parsed[2] AS CO2",
            "parsed[3] AS HC",
            "parsed[4] AS NOx",
            "parsed[5] AS PMx",
            "parsed[8] AS electricity",
            "parsed[9] AS fuel",
            "parsed[10] AS id",
            "parsed[12] AS noise",
            "parsed[15] AS speed",
            "parsed[16] AS type",
            "parsed[18] AS x",
            "parsed[19] AS y"
        )

@staticmethod
def process_region(stream:SparkSession, window_seconds = 15*60):
    parsed = stream.select( # type: ignore
        col("timestamp").cast("timestamp"),
        col("x").cast(DoubleType()),
        col("y").cast(DoubleType()),
    )
    return parsed

@staticmethod
def process_emissions(stream:SparkSession):
    parsed = stream.select( # type: ignore
        col("timestamp").cast("timestamp"),
        col("CO").cast(DoubleType()),
        col("CO2").cast(DoubleType()),
        col("HC").cast(DoubleType()),
        col("NOx").cast(DoubleType()),
        col("PMx").cast(DoubleType()),
        col("electricity").cast(DoubleType()),
        col("fuel").cast(DoubleType()),
        col("id").cast(DoubleType()),
        col("noise").cast(DoubleType()),
        col("speed").cast(DoubleType()),
        col("type")
    )
    return parsed
```

Modeli

- Za opis modela koristimo klase RegressionPredictor i ClusteringPredictor, koji su nasleđeni iz abstraktne klase Predictor
- Za regresiju koristimo GBTRegressor a za clusterizaciju KMeans

```
class Predictor:
    @abstractmethod
    def fit(ABC, X):
        pass
    @abstractmethod
    def predict(ABC, X):
        pass

class RegressionPredictor(Predictor):
    def __init__(self, target_label:str, features_label:str='features'):
        self.gbt = GBTRegressor(featuresCol=features_label, labelCol=target_label, seed = 1)
        self.model:GBTRegressionModel = None

    def fit(self, X):
        self.model = self.gbt.fit(X)

    def predict(self, X):
        return self.model.transform(X)

    def save(self, path):
        print("Saving to: ", path)
        self.model.write().overwrite().save(path)

    def load(self, path:str):
        print("Loading from: ", path)
        self.model = GBTRegressionModel()
        self.model.load(path)

    def name(self):
        return "GBT"

class ClusteringPredictor(Predictor):
    def __init__(self, k=2):
        self.kmeans = KMeans().setK(k).setSeed(1)
        self.model:KMeansModel = None

    def fit(self, X):
        self.model = self.kmeans.fit(X)

    def predict(self, X):
        return self.model.transform(X)

    def save(self, path):
        print("Saving to: ", path)
        self.model.write().overwrite().save(path)

    def load(self, path:str):
        print("Loading from: ", path)
        self.model = KMeansModel()
        self.model.load(path)

    def name(self):
        return "KM"
```

SPARK CONSUMER

Treniranje

- U funkciji train se vrši treniranje modela, pri čemu proveravamo da li je labela validna i biramo VectorAssembler-om podatke za trening. Nepostojeći podaci (NaN) se skipuju.
- Podaci se zatim fituju i istreniran model se čuva na disku

```
def train_model(self, pred: Predictor, X, col_list, output):
    if isinstance(pred, RegressionPredictor):
        if output not in ['CO','CO2','HC','NOx',"PMx"]:
            print("Output can only be one of: CO, CO2, HC, NOx or PMx")

    train_data = X.withColumn("timestamp_ms", unix_timestamp(col("timestamp")) * 1000)
    assembler = VectorAssembler(
        inputCols=col_list,
        outputCol="features",
        handleInvalid='skip'
    )
    assembled_data = assembler.transform(train_data)

    if isinstance(pred, RegressionPredictor):
        assembled_data = assembled_data.withColumnRenamed(output, "label")

    pred.fit(assembled_data)
    pred.save("./models/"+pred.name())
```

Predikcija

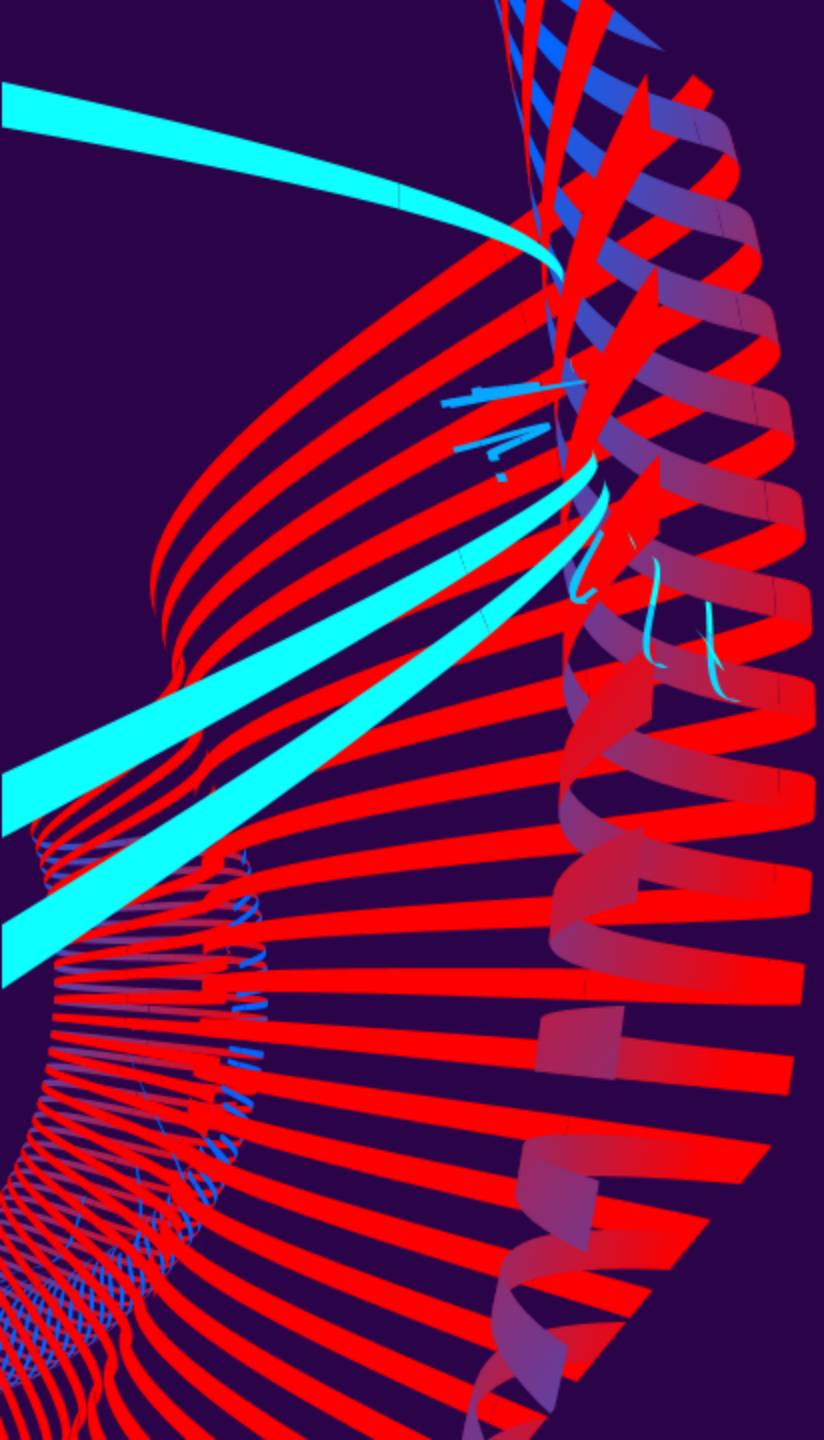
- U funkciji predict se vrši predikcija. Ukoliko model nije istreniran (None) prvo se učita sa diska.
- VectorAssembler vrši selekciju atributa za prediction a zatim se vrši predikcija. Nepostojeći podaci (NaN) se skipuju.

```
def predict(self, pred: Predictor, X, col_list, output):
    if pred.model is None:
        pred.load('./models/'+pred.name())
    if isinstance(pred, RegressionPredictor):
        if output not in ['CO','CO2','HC','NOx',"PMx"]:
            print("Output can only be one of: CO, CO2, HC, NOx or PMx")

    predict_data = X.withColumn("timestamp_ms", unix_timestamp(col("timestamp")) * 1000)

    try:
        assembler = VectorAssembler(
            inputCols=col_list,
            outputCol="features",
            handleInvalid='skip'
        )
        assembled_data = assembler.transform(predict_data)
    except Exception as e:
        print("Failed to assemble: ", e)
        exit(1)

    if isinstance(pred, RegressionPredictor):
        predictions = pred.predict(assembled_data).select(col("features").cast(StringType()), col("prediction"), col(output))
    else:
        predictions = pred.predict(assembled_data).select(col("features").cast(StringType()), col("prediction"))
    return predictions.withColumnRenamed("prediction", output+"_predicted")
```



DEPLOY I PERFORMANSE

PERFORMANSE

- Testiranje vršeno na Docker containeru.
- Program se izvršava na 1,2,4 ili 8 niti

SparkML

	1	2	3	AVG	STD
1	31.83417606	32.80637765	32.1150682	32.25187397	0.5003307361
2	31.06269884	31.95331788	31.83229518	31.61610397	0.4830678509
4	31.20582271	32.30233264	31.96209431	31.82341655	0.5612549764
8	31.99600387	32.22102952	32.17069578	32.12924306	0.118101154

HVALA NA PAŽNJI