Quora

July 11, 2017

1 The Problem

The problem is from Kaggle: https://www.kaggle.com/c/quora-question-pairs/data

In short, given a pair of questions. The goal is to predict if the pair of questions has the same meaning.

1.1 Data

The training data contains the following fields: * id - the id of a training set question pair * qid1, qid2 - unique ids of each question (only available in train.csv) * question1, question2 - the full text of each question * is_duplicate - the target variable, set to 1 if question1 and question2 have essentially the same meaning, and 0 otherwise.

1.2 The Solution

I treat this matching problem as a classification problem. More formally, I model the problem as below:

$$y = f(q1, q2) \ \ where \ \ y \in \{0, 1\}$$

q1 and *q2* are *question 1* and *question 2* respectively.

I used SparkML for this PoC. The training data file should be kept in HDFS to load it.

```
In [ ]: from pyspark.ml.feature import HashingTF, IDF, Tokenizer
        from pyspark.sql.types import ShortType
        from pyspark.ml.feature import StopWordsRemover
        from pyspark.ml.feature import RegexTokenizer
        from pyspark.ml.linalg import Vectors
        from pyspark.ml.feature import VectorAssembler
        from pyspark.sql.functions import length, udf, array, size
        from pyspark.sql.types import IntegerType
        from pyspark.ml.feature import StringIndexer, VectorIndexer
        from pyspark.ml.feature import StandardScaler
```

2 Preprocessing the Data

Clean the string data by tokenizing and removing stopwords.

```
In [ ]: def tokenize(p_df, in_column, out_column):
            """
            Tokenizes a column in a DataFrame.
            :param p_df: A DataFrame.
            :param in_column: Name of the input column.
            :param out_column: Name of the output column.
            :return: A DataFrame.
            """
            tokenizer = RegexTokenizer(inputCol=in_column, outputCol=out_column, pattern="\\W")
            return tokenizer.transform(p_df)

        def remove_stop_words(p_df, in_column, out_column):
            """
            Removes stop words from a column in a DataFrame. The column must be a list of words.
            :param p_df: A DataFrame.
            :param in_column: Name of the input column.
            :param out_column: Name of the output column.
            :return: A DataFrame.
            """
            remover = StopWordsRemover(inputCol=in_column, outputCol=out_column)
            return remover.transform(p_df)

        def clean_tokenize_remove_stopwords_quora(p_df, test_set=False):
            """
            Cleans, tokenizes, and removes stopwords from the quora dataset.
            :param p_df: A DataFrame.
            :param test_set: True or False for the quora data where the columns are different.
            :return: A DataFrame.
            """
            if not test_set:
                p_df = p_df.withColumnRenamed("is_duplicate", "label")
                p_df = p_df.withColumn("label", p_df["label"].cast(ShortType()))

            p_df = p_df.fillna("", ["question1", "question2"])
            if not test_set:
                p_df = p_df.fillna(0, ["label"])
            p_df = tokenize(p_df, "question1", "question1_words")
            p_df = remove_stop_words(p_df, "question1_words", "question1_meaningful_words")
            p_df = tokenize(p_df, "question2", "question2_words")
            p_df = remove_stop_words(p_df, "question2_words", "question2_meaningful_words")
            return p_df
```

3 Feature Engineering

I use TF-IDF features and some features derived from the question texts. The features derived from the texts are as below: * Lenght of question 1. * Length of question 2. * Difference between the length of question 1 and the length of question 2. * Number of words in question 1. * Number of words in question 2. * Number of common words in question1 and question 2.

```
In [ ]: def extract_tf_features(p_df, input_col, output_col):
            """
            Extracts TF features.
            :param p_df: A DataFrame.
            :param in_column: Name of the input column.
            :param out_column: Name of the output column.
            :return: A DataFrame.
            """
            hashingTF = HashingTF(inputCol=input_col, outputCol=output_col, numFeatures=3000)
            return hashingTF.transform(p_df)

        def extract_idf_features(p_df, input_col, output_col):
            """
            Extracts IDF features.
            :param p_df: A DataFrame.
            :param in_column: Name of the input column.
            :param out_column: Name of the output column.
            :return: A DataFrame.
            """
            idf = IDF(inputCol=input_col, outputCol=output_col)
            idfModel = idf.fit(p_df)
            return idfModel.transform(p_df)


        def tf_idf_features_quora(p_df):
            """
            Extracts TF-IDF features from quora dataset.
            :param p_df: A DataFrame.
            :return: A DataFrame.
            """
            tf_df = extract_tf_features(p_df, "question1_meaningful_words", "tf1")
            tf_df = extract_tf_features(tf_df, "question2_meaningful_words", "tf2")
            tf_idf_df = extract_idf_features(tf_df, "tf1", "tf-idf1")
            tf_idf_df = extract_idf_features(tf_idf_df, "tf2", "tf-idf2")
            assembler = VectorAssembler(
                inputCols=["tf-idf1", "tf-idf2"],
                outputCol="tf_idf_features"
            )
            return assembler.transform(tf_idf_df)

        def text_features(p_df):
            """
            Extracts features derived from the quora question texts.
            :param p_df: A DataFrame.
            :return: A DataFrame.
            """
```

```
    diff_len = udf(lambda arr: arr[0] - arr[1], IntegerType())
    common_words = udf(lambda arr: len(set(arr[0]).intersection(set(arr[1]))), IntegerTy
    unique_chars = udf(lambda s: len(''.join(set(s.replace(' ', '')))), IntegerType())


    p_df = p_df.withColumn("len_q1", length("question1")).withColumn("len_q2", length("q
    p_df = p_df.withColumn("diff_len", diff_len(array("len_q1", "len_q2")))
    p_df = p_df.withColumn("words_q1", size("question1_words")).withColumn("words_q2", s
    p_df = p_df.withColumn("common_words", common_words(array("question1_words", "questi
    p_df = p_df.withColumn(
        "unique_chars_q1", unique_chars("question1")
    ).withColumn("unique_chars_q2", unique_chars("question2"))

    assembler = VectorAssembler(
        inputCols=["len_q1", "len_q2", "diff_len", "words_q1", "words_q2", "common_words
        outputCol="text_features"
    )
    p_df = assembler.transform(p_df)
    return p_df
```

4 Load the Data and Extract Features

Loading the data and extracting the features we discussed before by calling the utility functions that we defined.

```
In [ ]: # Load the training data into a dataframe
        data = spark.read.format('json').load('train.jsonl')
        data = clean_tokenize_remove_stopwords_quora(data)

        # Get the tf-idf features
        data = tf_idf_features_quora(data)
        # Get the text features
        data = text_features(data)

        # combine all the features
        feature_assembler = VectorAssembler(
            inputCols=["tf_idf_features", "text_features"],
            outputCol="combined_features"
        )
        data = feature_assembler.transform(data)


        # Normalizing each feature to have unit standard deviation
        scaler = StandardScaler(inputCol="combined_features", outputCol="features",
                                withStd=True, withMean=False)
        scalerModel = scaler.fit(data)
        # Normalize each feature to have unit standard deviation.
```

```
data = scalerModel.transform(data)


# Index labels, adding metadata to the label column.
# Fit on whole dataset to include all labels in index.
label_indexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
# Automatically identify categorical features, and index them.
feature_indexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCat

training_df, test_df = data.randomSplit([0.8, 0.2])
training_df.cache()
test_df.cache()
```

5 Models

I experimented with Logistic Regression, Decision Tree, and Random Forest. But first I am defining a utility function to print the evaluation metrics.

6 Utility Functions

In [ ]:
```
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

def print_evaluation_metrics(model, test_df, labelCol="label", featuresCol="features"):
    """
    Prints evaluation metrics.
    :param model: Used model.
    :param test_df: dataframe containing test data.
    :param labelCol: label column.
    :param featuresCol: features column.
    :return: A DataFrame.
    """
    predictions = model.transform(test_df)


    # Select (prediction, true label) and compute test error
    evaluator = MulticlassClassificationEvaluator(
        labelCol=labelCol, predictionCol="prediction",)
    accuracy = evaluator.evaluate(predictions, {evaluator.metricName: "accuracy"})
    f1 = evaluator.evaluate(predictions, {evaluator.metricName: "f1"})
    weighted_precision = evaluator.evaluate(predictions, {evaluator.metricName: "weighte
    weighted_recall = evaluator.evaluate(predictions, {evaluator.metricName: "weightedRe
    print "Accuracy:", accuracy
    print "f1:", f1
    print "Precision:", weighted_precision
    print "Recall:", weighted_recall
```

6.1 Logistic Regression

I used 10 fold cross validation to select the parameters for Logistic Regression.

```
In [ ]: from pyspark.ml.classification import LogisticRegression
        from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
        from pyspark.ml.evaluation import BinaryClassificationEvaluator
        lr = LogisticRegression(maxIter=100, elasticNetParam=0.8)


        paramGrid = ParamGridBuilder() \
            .addGrid(lr.regParam, [0, 0.001, 0.003, 0.01, 0.03, 0.1, 0.3, 1, 3, 10]) \
            .addGrid(lr.elasticNetParam, [0, 0.01, 0.03, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8,
            .build()

        crossval = CrossValidator(estimator=lr,
                                  estimatorParamMaps=paramGrid,
                                  evaluator=BinaryClassificationEvaluator(),
                                  numFolds=10)  # 10 fold cross validation

        # Fit the model
        lrModel = lr.fit(training_df)
```

```
In [ ]: print_evaluation_metrics(lrModel, test_df, labelCol="label", featuresCol="features")
```

Logictic Regression performs as below: * Accuracy: 0.756172724449 * f1: 0.753834390431 * Precision: 0.752930056979 * Recall: 0.756172724449

6.2 Decision Tree

```
In [ ]: from pyspark.ml import Pipeline
        from pyspark.ml.classification import DecisionTreeClassifier
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator


        # Train a DecisionTree model.
        dt = DecisionTreeClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures")

        # Chain indexers and tree in a Pipeline
        pipeline = Pipeline(stages=[label_indexer, feature_indexer, dt])

        # Train model.  This also runs the indexers.
        model = pipeline.fit(training_df)
```

```
In [ ]: print_evaluation_metrics(model, test_df, labelCol="indexedLabel", featuresCol="indexedFe
```

The accuracy and f1-socre go down with Desicion Tree: * Accuracy: 0.674420627524 * f1: 0.665704726497 * Precision: 0.664141841759 * Recall: 0.674420627524

6.3 Random Forest

```
In [ ]: from pyspark.ml import Pipeline
        from pyspark.ml.classification import RandomForestClassifier
        from pyspark.ml.feature import StringIndexer, VectorIndexer
        from pyspark.ml.evaluation import MulticlassClassificationEvaluator


        # Train a RandomForest model.
        rf = RandomForestClassifier(labelCol="indexedLabel", featuresCol="indexedFeatures", numT

        # Chain indexers and forest in a Pipeline
        pipeline = Pipeline(stages=[label_indexer, feature_indexer, rf])

        # Train model.  This also runs the indexers.
        model = pipeline.fit(training_df)
```

```
In [ ]: print_evaluation_metrics(model, test_df, labelCol="indexedLabel", featuresCol="indexedFe
```

Random forest gives us the worst perfomance: * Accuracy: 0.632382727555 * f1: 0.491917348606 * Precision: 0.755623695496 * Recall: 0.632382727555

7 Conclusion

Logistic Regression gives us the best performance with accuracy 0.756172724449 and f1-score 0.753834390431. It would be interesting to see how this approach will peform if we use more semantic features such as Word2Vec and LSA.