Homework 6: Home Price Prediction with Regression

一、模型建立与数据预处理

首先我们要对数据进行预处理：把Nan的数据进行替换，对于类别数据进行one-hot编码，数值类数据要进行标准化

第一步：将feature进行分类，分成类别型和数值型这两类

```
categorical_features = train.select_dtypes(include=["object"]).columns
numerical_features = train.select_dtypes(exclude = ["object"]).columns
numerical_features = numerical_features.drop(["SalePrice", "Id"])
train_cat = train[categorical_features]
train_num = train[numerical_features]
```

通过`select_dtypes`函数对feature进行筛选。`include=["object"]`代表了类别型的feature，反之`exclude = ["object"]`就代表了数值型的feature。然后就可以很容易的得到相应feature分开的数据`train_cat`和`train_num`。

第二步：把Nan的数据用中位数进行替换

通过`train_num.isnull().sum()`，可以得到各个属性出现nan的次数，结果如下：

```
LotFrontage     227
GarageYrBlt      78
MasVnrArea       15
BsmtHalfBath      2
BsmtFullBath      2
GarageArea        1
GarageCars        1
TotalBsmtSF       1
BsmtUnfSF         1
BsmtFinSF2        1
BsmtFinSF1        1
```

我们把Nan的数据用对应feature的中位数进行替换：

```
train_num = train_num.fillna(train_num.median())
```

第三步：对于类别数据进行one-hot编码

MSZoning_C (all)	MSZoning_FV	MSZoning_RH	MSZoning_RL	MSZoning_RM
0	0	1	0	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0
0	0	0	1	0

如上图所示：

我们将一个feature转变成多个feature的组合，即feature的每个类别都产生一个新的feature。

对于原feature `MSZoning`，使用五个新feature表示。如果原来是`C(all)`就用`10000`来表示；`FV`用`01000`表示；以此类推，可以得到五个类别的表示。实现代码如下：

```
train_cat = pd.get_dummies(train_cat)
```

第四步：数值类数据进行标准化

标准化就是把数据变成标准正态分布，方便后续训练。代码如下：

```
# mean normalization
train_num_normalized = (train_num-train_num.mean()) / train_num.std()
```

第五步：数据整合，label生成

将上面处理好的两类数据进行整合得到最后的数据：

```
train_clean = pd.concat([train_cat, train_num_normalized], axis=1)
```

提取数据中`SalePrice`属性，然后进行对数化，作为训练的标签：

```
train.SalePrice = np.log(train.SalePrice)
```

二、算法实现

1. PCA算法实现

参考github中的一份代码https://github.com/anujdutt9/BigData-and-Machine-Learning/blob/master/Apache%20Spark%20Machine%20Learning%20using%20Scala/PCA/PCAExercise.scala

经过修改后，PCA部分的代码如下：

```
import org.apache.spark.sql.SparkSession
import org.apache.log4j._
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.feature.PCA
// Import Vectors from ml.linalg
import org.apache.spark.ml.linalg.Vectors

object PCA_Test{
    def main(args:Array[String]){
        Logger.getLogger("org").setLevel(Level.ERROR)
        // Use Spark to read in the Cancer_Data file.
        val spark = SparkSession.builder().appName("PCA").getOrCreate()

        val data =
spark.read.option("Header","true").option("inferSchema","true").format("csv").lo
ad("train_clean.csv")
        // Print the Schema of the data
        data.printSchema
        // Import PCA, VectorAssembler and StandardScaler from ml.feature
```

```
        // Use VectorAssembler to convert the input columns of the cancer data
        // to a single output column of an array called "features"
        // Set the input columns from which we are supposed to read the values.
        // Call this new object assembler.

        // Since there are so many columns, you may find this line useful
        // to just pass in to setInputCols

        val colnames = (Array("MSZoning_C
(all)","MSZoning_FV","MSZoning_RH",...,"MoSold","YrSold"))

        val assembler = new
VectorAssembler().setInputCols(colnames).setOutputCol("features")

        // Use the assembler to transform our DataFrame to a single column:
features
        val output = assembler.transform(data).select("features")
        // Often its a good idea to normalize each feature to have unit standard
        // deviation and/or zero mean","vwhen using PCA.
        // This is essentially a pre-step to PCA"," but its not always
necessary.
        // Look at the ml.feature documentation and figure out how to
standardize
        // the cancer data set. Refer to the solutions for hints if you get
stuck.

        // Use StandardScaler on the data
        // Create a new StandardScaler() object called scaler
        // Set the input to the features column and the ouput to a column called
        // scaledFeatures

        val scalar = (new
StandardScaler().setInputCol("features").setOutputCol("scaledFeatures").setWithS
td(true).setWithMean(false))
        // Compute summary statistics by fitting the StandardScaler.
        // Basically create a new object called scalerModel by using
scaler.fit()
        // on the output of the VectorAssembler
        val scalarModel = scalar.fit(output)
        // Normalize each feature to have unit standard deviation.
        // Use transform() off of this scalerModel object to create your
scaledData
        val scaledData = scalarModel.transform(output)
        // Now its time to use PCA to reduce the features to some principal
components

        // Create a new PCA() object that will take in the scaledFeatures
        // and output the pcs features"," use 4 principal components
        // Then fit this to the scaledData

        val pca = (new PCA()
                    .setInputCol("scaledFeatures")
                    .setOutputCol("pcaFeatures")
                    .setK(10)
                    .fit(scaledData))
        // Once your pca has been created and fit"," transform the scaledData
        // Call this new dataframe pcaDF
        val pcaDF = pca.transform(scaledData)
```

```
        // Show the new pcaFeatures
        val results = pcaDF.select("pcaFeatures")
        results.show()
        // Use .head() to confirm that your output column Array of pcaFeatures
        // only has 4 principal components

        results.write.mode("overwrite").json("output")
    }
}
```

以上代码省略了冗长的特征名称，其主要做了以下这些事：1.读入csv文件，2.将所有特征并为一个feature，3.使用StandardScaler将所有特征归一化，4.使用PCA对所有数据降维，降为10个主要特征，5.将结果显示和保存。

在编译和运行代码时，所用到的依赖和上周的作业一样，我们同样使用了sbt来帮助编译，在此不多赘述。

运行代码时，程序先输出了所有的特征属性：

```
root
 |-- MSZoning_C (all): integer (nullable = true)
 |-- MSZoning_FV: integer (nullable = true)
 |-- MSZoning_RH: integer (nullable = true)
 |-- MSZoning_RL: integer (nullable = true)
 |-- MSZoning_RM: integer (nullable = true)
 |-- Street_Grvl: integer (nullable = true)
 |-- Street_Pave: integer (nullable = true)
 |-- Alley_Grvl: integer (nullable = true)
 |-- Alley_Pave: integer (nullable = true)
 |-- LotShape_IR1: integer (nullable = true)
 |-- LotShape_IR2: integer (nullable = true)
 |-- LotShape_IR3: integer (nullable = true)
 |-- LotShape_Reg: integer (nullable = true)
 |-- LandContour_Bnk: integer (nullable = true)
 |-- LandContour_HLS: integer (nullable = true)
 |-- LandContour_Low: integer (nullable = true)
 |-- LandContour_Lvl: integer (nullable = true)
 |-- Utilities_AllPub: integer (nullable = true)
 |-- Utilities_NoSeWa: integer (nullable = true)
 |-- LotConfig_Corner: integer (nullable = true)
 |-- LotConfig_CulDSac: integer (nullable = true)
 |-- LotConfig_FR2: integer (nullable = true)
 |-- LotConfig_FR3: integer (nullable = true)
 |-- LotConfig_Inside: integer (nullable = true)
 |-- LandSlope_Gtl: integer (nullable = true)
 |-- LandSlope_Mod: integer (nullable = true)
 |-- LandSlope_Sev: integer (nullable = true)
 |-- Neighborhood_Blmngtn: integer (nullable = true)
 |-- Neighborhood_Blueste: integer (nullable = true)
 |-- Neighborhood_BrDale: integer (nullable = true)
 |-- Neighborhood_BrkSide: integer (nullable = true)
 |-- Neighborhood_ClearCr: integer (nullable = true)
 |-- Neighborhood_CollgCr: integer (nullable = true)
 |-- Neighborhood_Crawfor: integer (nullable = true)
 |-- Neighborhood_Edwards: integer (nullable = true)
 |-- Neighborhood_Gilbert: integer (nullable = true)
 |-- Neighborhood_IDOTRR: integer (nullable = true)
 |-- Neighborhood_MeadowV: integer (nullable = true)
 |-- Neighborhood_Mitchel: integer (nullable = true)
 |-- Neighborhood_NAmes: integer (nullable = true)
 |-- Neighborhood_NPkVill: integer (nullable = true)
 |-- Neighborhood_NWAmes: integer (nullable = true)
 |-- Neighborhood_NoRidge: integer (nullable = true)
 |-- Neighborhood_NridgHt: integer (nullable = true)
 |-- Neighborhood_OldTown: integer (nullable = true)
 |-- Neighborhood_SWISU: integer (nullable = true)
 |-- Neighborhood_Sawyer: integer (nullable = true)
 |-- Neighborhood_SawyerW: integer (nullable = true)
 |-- Neighborhood_Somerst: integer (nullable = true)
 |-- Neighborhood_StoneBr: integer (nullable = true)
 |-- Neighborhood_Timber: integer (nullable = true)
```

随后简单的输出了降维后，各个主成分的大小，程序只显示了第一个成分。

```
+--------------------+
|         pcaFeatures|
+--------------------+
|[7.47509290109241...|
|[3.31142489287071...|
|[8.06178996758529...|
|[1.14194742373612...|
|[9.6016007509898,...|
|[3.67926096003885...|
|[8.98775465292387...|
|[4.77204777788009...|
|[-3.5371023479271...|
|[-1.1637520892243...|
|[0.46385497362644...|
|[10.6089898189473...|
|[-0.3564857708134...|
|[9.76512319519041...|
|[1.24517095551337...|
|[-1.4342520499326...|
|[2.39943633198688...|
|[-1.0251343370873...|
|[3.98178425884629...|
|[-0.5951094450824...|
+--------------------+
only showing top 20 rows
```

随后找到输出文件`$SPARK_HOME/bin/output`，如下：

```
vim part-00000-14d82f6d-89cf-42c4-99b0-fa7df049c2df-c000.json
{"pcaFeatures":{"type":1,"values":[7.475092901092416,1.0448464938270383,-8.85341
8243158957,3.80648737455011,-1.4576022282407077,-0.3189067054253557,-3.958731948
7769053,-1.104298979918653,5.205492297963189,-1.4558465493655617]}}
{"pcaFeatures":{"type":1,"values":[3.31142489287071,5.497819665373972,-6.6704732
88877777,2.678089370147132,-3.4133772808622114,-0.46571756648158147,-2.123541595
166727,-1.0280412581920753,5.240901892663381,-1.9044298209591097]}}
{"pcaFeatures":{"type":1,"values":[8.061789967585296,1.5413555147125693,-8.05853
2051018062,4.071089452144886,-1.0560418672945653,-1.1393852569312577,-3.94393138
76998634,-0.9676998999961167,5.266134604001188,-1.1630031773115288]}}
{"pcaFeatures":{"type":1,"values":[1.1419474237361282,1.708946881095822,-5.63952
1350330838,5.845213705765398,-4.677407131365809,-0.21012572078209404,-1.87022729
55049335,-0.9851072044989209,4.330418406811753,-3.5711533357880607]}}
{"pcaFeatures":{"type":1,"values":[9.6016007509898,1.6289027264060862,-6.1672521
9933761,5.612659440422845,-0.5579263116387105,-0.07948422929639035,-4.4141045699
88527,-0.3751551323765466З,5.78096710703321,-0.6260974388301096]}}
{"pcaFeatures":{"type":1,"values":[3.6792609600388535,3.6314199306604316,-8.5363
09142685496,2.8717156543690687,-2.892048253710134,-2.1567756542872405,-2.6234113
332665756,-1.459195087948936,4.370146331133728,-0.26852787090609254]}}
{"pcaFeatures":{"type":1,"values":[8.987754652923877,1.6426918837662603,-6.29970
8220936043,0.2794554460676262,-4.542198347331853,0.29022885636624757,-2.78069157
36662345,-1.3719114867401143,5.403021516365609,-1.9188069207850473]}}
@
@
-- INSERT --                                              1,2           Top
```

在之后的线性和决策树模型中，就可以处理以上得到的主成分进行预测。

2. 线性模型

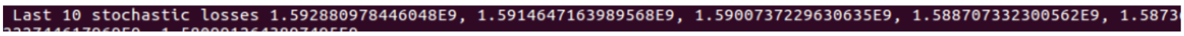
线性模型的代码如下，和上一次作业用到的是同一份代码，只稍作修改：

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LinearRegressionModel
```

```scala
object linear {
  def main(args: Array[String]) {
    val inputPath = args(0)
    val iterations = args(1).toInt

    val spark = SparkSession
      .builder
      .appName("linear")
      .getOrCreate()

    // Load and parse the data
    val data = spark.read.textFile(inputPath).rdd
    val parsedData = data.map { line =>
      val parts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split('
').map(_.toDouble)))
    }.cache()

    // Building the model
    val step_size = 0.01
    val model = LinearRegressionWithSGD.train(parsedData, iterations,step_size)

    // Evaluate model on training examples and compute training error
    val valuesAndPreds = parsedData.map { point =>
      val prediction = model.predict(point.features)
      (point.label, prediction)
    }
    // valuesAndPreds.show()
    val MSE = valuesAndPreds.map{case(v, p) => math.pow((v - p), 2)}.mean()
    println("training Mean Squared Error = " + MSE)
    spark.stop()
  }
}
```

将迭代次数设置为100，运行结果如下：

可以看到最后几次迭代的损失函数基本不变，说明算法收敛。



我们程序中计算的是MSE均方误差，约为3.1×10^9，计算RMSE均方根误差，则约为5.5×10^4。

3.决策树模型

我们借鉴了spark官网的决策树回归代码，修改输入方式后如下：

```scala
import org.apache.spark.sql.SparkSession
```

```scala
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint

object dtree{
  def main(args: Array[String]){
    val spark = SparkSession
      .builder
      .appName("decision_tree")
      .getOrCreate()

    // Load and parse the data file.
    val inputPath = "Data.txt"
    val data = spark.read.textFile(inputPath).rdd
    val parsedData = data.map { line =>
      val parts = line.split(',')
      LabeledPoint(parts(0).toDouble, Vectors.dense(parts(1).split('
').map(_.toDouble)))
    }.cache()

    val splits = parsedData.randomSplit(Array(0.7, 0.3))
    val (trainingData, testData) = (splits(0), splits(1))


    // Split the data into training and test sets (30% held out for testing)


    val categoricalFeaturesInfo = Map[Int, Int]()
    val impurity = "variance"
    val maxDepth = 5
    val maxBins = 32

    val model = DecisionTree.trainRegressor(trainingData,
categoricalFeaturesInfo, impurity,
      maxDepth, maxBins)

    // Evaluate model on test instances and compute test error
    val labelsAndPredictions = testData.map { point =>
      val prediction = model.predict(point.features)
      (point.label, prediction)
    }
    val testMSE = labelsAndPredictions.map{ case (v, p) => math.pow(v - p, 2)
}.mean()
    println("Test Mean Squared Error = " + testMSE)
    println("Learned regression tree model:\n" + model.toDebugString)

    // Save and load model
    // model.save(spark, "target/tmp/myDecisionTreeRegressionModel")
    // val sameModel = DecisionTreeModel.load(spark,
"target/tmp/myDecisionTreeRegressionModel")
    spark.stop()
  }

}
```

程序运行的结果如下：

```
19/12/13 20:55:52 INFO DAGScheduler: Job 8 finished: mean at DT.scala:42, took 0.162958 s
Test Mean Squared Error = 2.8465076213121223E9
Learned regression tree model:
DecisionTreeModel regressor of depth 5 with 57 nodes
  If (feature 0 <= 8.071219771268227)
   If (feature 0 <= 1.369455891158448)
    If (feature 0 <= -2.593343885408971)
     If (feature 0 <= -5.401266389377938)
      If (feature 5 <= 1.3924462385100371)
       Predict: 67104.82608695653
      Else (feature 5 > 1.3924462385100371)
       Predict: 103905.55555555556
     Else (feature 0 > -5.401266389377938)
      If (feature 2 <= -2.234561702440478)
       Predict: 103817.38983050847
      Else (feature 2 > -2.234561702440478)
       Predict: 156380.0
    Else (feature 0 > -2.593343885408971)
     If (feature 2 <= -4.22899802565431)
      If (feature 2 <= -7.810279408001384)
       Predict: 114949.91262135922
      Else (feature 2 > -7.810279408001384)
       Predict: 135934.3686868687
     Else (feature 2 > -4.22899802565431)
      If (feature 3 <= 6.925705600323628)
       Predict: 181425.0
      Else (feature 3 > 6.925705600323628)
       Predict: 260989.5
   Else (feature 0 > 1.369455891158448)
    If (feature 2 <= -4.22899802565431)
     If (feature 0 <= 5.587622368868127)
      If (feature 2 <= -6.795116755005289)
       Predict: 154782.29090909092
      Else (feature 2 > -6.795116755005289)
       Predict: 183682.54716981133
     Else (feature 0 > 5.587622368868127)
      If (feature 2 <= -7.149099513627264)
       Predict: 195031.40714285715
      Else (feature 2 > -7.149099513627264)
       Predict: 232564.08333333334
    Else (feature 2 > -4.22899802565431)
     If (feature 8 <= 4.609313029475614)
      If (feature 2 <= -2.234561702440478)
       Predict: 196343.75
      Else (feature 2 > -2.234561702440478)
       Predict: 251763.75
     Else (feature 8 > 4.609313029475614)
      If (feature 7 <= -2.917370448003097)
       Predict: 425000.0
      Else (feature 7 > -2.917370448003097)
       Predict: 287830.0
```

从上图可以窥见训练得到的树模型，以及均方误差$MSE = 2.8 \times 10^9$，开根号后$RMSE = 5.3 \times 10^4$。决策树模型的误差略小于线性模型。

三、遇到的问题

1.pca输出csv报错

MLlib的PCA库有类似pandas的数据结构pcadf，但是它不支持复杂结构的csv输出，我们尝试了把每个feature转换成str输出，但输出结果也不太对，最终把它用json格式输出。

2.线性模型的参数问题

由于MLlib是使用SGD方法对线性模型进行优化的，它的参数设置有一定的技巧，尤其是step_size要设置合理，不然可能会导致模型无法收敛。我们查阅了stackoverflow的提问，有提到以下的步长选取方法，因为我们的最大特征约为10，总共有10个特征，所以L约为$10^2/10 = 10$步长应该为$1/2 \times L = 0.05$，最终我们选择了0.01。

> The step size should be smaller than 1 over the Lipschitz constant `L`. For quadratic loss and GD, the best convergence happens at `stepSize = 1/(2L)`. Spark has a `(1/n)` multiplier on the loss function.
>
> Let's say you have `n = 5` data points and the largest feature value is `1500`. So `L = 1500 * 1500 / 5`. The best convergence happens at `stepSize = 1/(2L) = 10 / (1500 ^ 2)`.

3.MLlib的数据格式

通常MLlib处理的格式要进行一定转化后再读取，比如我们的数据处理格式为

```
label,feature1 feature2 feature3 feature4 ...
```

4.spark无法解析带小数的特征名称

将输入特征和表中特征名称中的小数点改为下划线。比如原特征是HouseStyle_1.5Fin，改成了HouseStyle_1_5Fin。

四、组员分工

胡晨旭：数据预处理以及算法搭建

王宇琪：算法模型搭建以及数据处理

张智为：代码运行以及报告整合