華東理工大學

模式识别大作业

题　　目　汽车投保风险预测

学　　院　信息科学与工程学院

专　　业　控制科学与工程

组　　员　庄 屹

指导教师　赵海涛

完成日期：　2019 年 11 月 21 日

模式识别作业报告——汽车投保风险预测

组员：庄屹

在第一个学习单元中，选修了赵海涛老师的模式识别原理与应用这门课程，在赵老师的悉心的指导下，对模式识别与机器学习中的部分原理，以及相关的基础算法有了一定的理解，为了能够更好地将学到的原理方法运用到实际问题的解决中中，从而更有体会地掌握知识，进行了实际问题的实践。

1 汽车投保风险预测简介

数据来源于 Lintcode 中的 Problem List。针对某保险公司销售一种汽车保险，需要对汽车状态进行评估。现需要设计一个算法模型，可以根据汽车的各项指标对汽车的投保风险进行打分。投保风险是从 0 到 70 的正整数，数值越大代表风险越高。

数据文件 train.csv 和 test.csv 包含多辆汽车的信息。其中训练数据共 32000 条，测试数据共 8000 条。每辆汽车有如下信息：

Id: 每个汽车的唯一 id

Score: 汽车的风险数值

Col_1 ~ Col_32: 每个汽车的 32 个特征。

训练数据集(train.csv)包含 34 列，分别对应上述信息。

测试数据集(test.csv)包含 33 列，不包含 Score 信息。

通过数据集给定汽车的各项性能指标，设计算法对汽车的投保风险进行打分。最后，使用 RMSE (Root Mean Square Error)作为评价指标。

2 整体解决方案

这次通过题目的描述，可以简单看出是一个典型的回归问题。针对回归问题，解决的方案，可以使用传统的回归模型，如 SVR(支持向量回归)，CART(分类与回归树)等。或者使用集成学习的算法将多个简单模型集成为一个复杂模型，如 Random Forest(随机森林)，Adaboost 等，在使用集成学习(Ensemble Learning)的方法时，要注意引入正则化项，防止模型过拟合。以及利用深度学习中的神经网络结构挖掘特征之间的关系。本作业中将分别利用这些方法进行实验比较。

2.1 数据结构分析

首先观察原始训练数据的结构，原始数据的分类如表 1 所示。

表 1 原始数据结果

Id	Col_i	Col_29	Col_30	Col_31	Col_32	Score
1	...	b	15	15	b	4
2	...	b	15	3	b	2
3	...	d	10	18	b	4
4	...	b	15	18	b	1

表中的数据只是截取了原始数据中的一部分。

第 1 列数据表示的是数据的编号（Id），不需要用到，可以在数据预处理过程中删除了这列数据；

第 2 到第 33 列分别为汽车的 32 个特征。对应的 index 分别用 Col_i 表示（i 表示为第 i 个特征）。这些特征中，有一部分特征为整型数值，还用一部分为类别型。第 3，4，8，9，11，12，15，17，20，22，24，25，27，28，29，32 个特征为类别型，每个类别分别用 a-z 的字母代表其不同的分类，且每个特征中的类别总数不等，这对之后的特征编码产生了一定的影响。

第 34 列为最终的风险数值，为整型，仅在训练数据中有这一列。而在测试数据中不存在这一列。

2.2 数据预处理

2.2.1 传统编码

数据的预处理主要是考虑针对类别型的数据进行编码。常见的编码形式主要有标签编码（Label Encoding）以及独立热编码（One-Hot Encoding）。

标签编码是将原始的类别型变量转换成连续的数值型变量。即是对不连续的数字或者文本进行编号。而标签编码的场景限制很多，并且解释性较差。比如有[a,b,c]，我们把其转换为[1,2,3]。这里就产生了一个奇怪的现象：a 和 c 的平均值是 b。这对于类别型的变量来说就几乎没有什么可解释性，因此，标签编码没有被广泛使用。部分标签编码后的数据如表 2 所示

表 2 标签编码后数据结果

Id	1_Col_3	1_Col_4	1_Col_8	1_Col_9	1_Col_11	Score
1	0	1	4	1	2	4
2	0	0	2	1	3	2
3	0	0	4	1	3	4
4	0	0	4	1	0	1

独立热编码是目前大部分算法所用到的编码方式。目前大部分的算法都是基于欧式空间中的度量来进行计算的，为了使非偏序关系的变量取值不具有偏序

性，将离散特征的取值扩展到了欧式空间，离散特征的某个取值就对应欧式空间的某个点。将离散型特征使用独热编码，会让特征之间的距离计算更加合理。离散特征进行编码后，每一维度的特征都可以看做是连续的特征。就可以跟对连续型特征的归一化方法一样，对每一维特征进行归一化。对于原始数据的类别变量进行独立热编码后的部分数据结果如表 3 所示。

表 3 独立热编码后数据结果

Id	oh_Col_29_a	oh_Col_29_b	oh_Col_29_c	oh_Col_29_d	oh_Col_32_a	Score
1	0	1	0	0	0	4
2	0	1	0	0	0	2
3	0	0	0	1	0	4
4	0	1	0	0	0	1

独热编码解决了分类器不好处理属性数据的问题，在一定程度上也起到了扩充特征的作用。它的值只有 0 和 1，不同的类型存储在垂直的空间。但是独立热编码同样存在一些限制，当类别的数量很多时，特征空间会变得非常大。在这种情况下，一般可以用 PCA 来减少维度。

2.2.2 均值编码

下面介绍一种新的编码方式：均值编码（Mean Encoding）。也可以被叫做是期望编码（likelihood encoding）、或者是目标编码（target encoding）如果某一个特征是定性的（categorical），而这个特征的可能值非常多（高基数），那么平均数编码是一种高效的编码方式。在实际应用中，这类特征工程能极大提升模型的性能。

基本思想是：

假设在分类问题中，目标 y 一共有 C 个不同类别，具体的一个类别用 target 表示；某一个定性特征 variable 一共有 K 个不同类别，具体的一个类别用 k 表示。

则先验概率估计可以用式（1）表示：

$$\hat{P}(y = t\arg et) \tag{1}$$

将 variable 中的每一个 k，都表示为（估算的）它所对应的目标 y 值概率,如式（2）所示：

$$\hat{P}(t\arg et = y \mid \mathrm{var}\, iable = k) \tag{2}$$

得到了先验概率估计式（1）和后验概率估计式（2）。最终编码所使用的概率估算，应当是先验概率与后验概率的一个凸组合，引入先验概率的权重 λ：

$$\hat{P} = \lambda * \hat{P}(y = t\arg et) + (1-\lambda) * \hat{P}(t\arg et = y \mid \mathrm{var}\, iable = k) \tag{3}$$

λ 应该具有以下的特性：如果测试集中出现了新的特征类别（未在训练集中

出现），那么为 1。一个特征类别在训练集内出现的次数越多，后验概率的可信度越高，其权重也越大。需要定义一个权重函数，输入是特征类别在训练集中出现的次数 n，输出是对于这个特征类别的先验概率的权重 λ。假设一个特征类别的出现次数为 n，常见的权重函数：

$$\lambda(n)=\frac{1}{1+e^{(n-k)/f}} \tag{4}$$

对比前两种的编码方式标签编码高基数定性特征，虽然只需要一列，但是每个自然数都具有不同的重要意义，对于预测结果而言线性不可分。使用简单模型，容易欠拟合，无法完全捕获不同类别之间的区别；使用复杂模型，容易在其他地方过拟合。

而使用独立热编码高基数定性特征，会产生较大的稀疏矩阵，易消耗大量内存和训练时间。具体代码如下：

```
    def impact_coding(self, data, feature, target='Score'):
        n_folds = 20
        n_inner_folds = 10
        impact_coded = pd.Series()
        oof_default_mean = data[target].mean()  # Gobal mean to use by default (you could further tune this)
        kf = KFold(n_splits=n_folds, shuffle=True)
        oof_mean_cv = pd.DataFrame()
        split = 0
        for infold, oof in kf.split(data[feature]):
            impact_coded_cv = pd.Series()
            kf_inner = KFold(n_splits=n_inner_folds, shuffle=True)
            inner_split = 0
            inner_oof_mean_cv = pd.DataFrame()
            oof_default_inner_mean = data.iloc[infold][target].mean()
            for infold_inner, oof_inner in kf_inner.split(data.iloc[infold]):
                # The mean to apply to the inner oof split (a 1/n_folds % based on the rest)
                oof_mean = data.iloc[infold_inner].groupby(by=feature)[target].mean()
                impact_coded_cv = impact_coded_cv.append(data.iloc[infold].apply(lambda x: oof_mean[x[feature]] if x[feature] in oof_mean.index else oof_default_inner_mean, axis=1))

                # Also populate mapping (this has all group -> mean for all inner CV folds)
                inner_oof_mean_cv = inner_oof_mean_cv.join(pd.DataFrame(oof_mean), rsuffix=inner_split, how='outer')
                inner_oof_mean_cv.fillna(value=oof_default_inner_mean, inplace=True)
                inner_split += 1
            # Also populate mapping
            oof_mean_cv = oof_mean_cv.join(pd.DataFrame(inner_oof_mean_cv), rsuffix=split, how='outer')
            oof_mean_cv.fillna(value=oof_default_mean, inplace=True)
```

```
            split += 1
            impact_coded = impact_coded.append(data.iloc[oof].apply(
                lambda x: inner_oof_mean_cv.loc[x[feature]].mean()
                if x[feature] in inner_oof_mean_cv.index
                else oof_default_mean
                , axis=1))
        return impact_coded, oof_mean_cv.mean(axis=1), oof_default_mean
```

算法思路：

把 train data 划分为 20-folds

将每一个 infold 再次划分为 10-folds

计算 10-folds 的 inner out of folds 值

对 10 个 inner out of folds 值取平均，得到 inner_oof_mean

计算 oof_mean

将 train data 的 oof_mean 映射到 test data 完成编码

如果 fit 时使用了全部的数据，transform 时也使用了全部数据，那么之后的机器学习模型会产生过拟合。因此，我们需要将数据分层分为 n_splits 个 fold，每一个 fold 的数据都是利用剩下的(n_splits - 1)个 fold 得出的统计数据进行转换。n_splits 越大，编码的精度越高，但也更消耗内存和运算时间。均值编码后的部分数据结果如表 4 所示。

表 4 均值编码后数据结果

Id	m_Col_25	m_Col_27	m_Col_28	m_Col_29	m_Col_32	Score
1	3.955855688	3.961216543	4.034264005	3.873077757	4.014574356	4
2	4.0102357	4.611160969	4.034268478	3.873103029	4.014223532	2
3	4.010213 84	4.611788868	4.034264005	3.957736422	4.014223532	4
4	4.010271379	3.296815 38	4.034075869	3.873035828	4.014223532	1

2.3 特征选取

在实际进行模型回归之前，首先需要确定一下各个特征与实际的结果之间是否存在相关关系。倘若特征与实际结果之间随机性非常强，不存在显著关系，那么对于模型来说可能一定程度上会影响实际的回归效果。本次实验运用了 catboost 对于特征的重要属性进行探究。对于 catboost 的计算原理不再进行赘述。具体代码如下：

```
    def my_cat(self):
        categorical_features_indices = np.where(self.training_x.dtypes != np.int64)[0]
```

```
        model = CatBoostRegressor(iterations=5000, cat_features=categorical_features_indices,
                                  learning_rate=0.01, loss_function='RMSE',
                                  logging_level='Verbose')
        model.fit(self.training_x, self.training_y, eval_set=(self.validation_x, self.validation_y),
plot=True)
        fea_ = model.feature_importances_
        fea_name = model.feature_names_
        plt.figure(figsize=(10, 10))
        plt.barh(fea_name, fea_, height=0.5)
        plt.show()
```

最终，绘制得到每个特征对于实际预测结果的重要属性值。如图 1 所示：

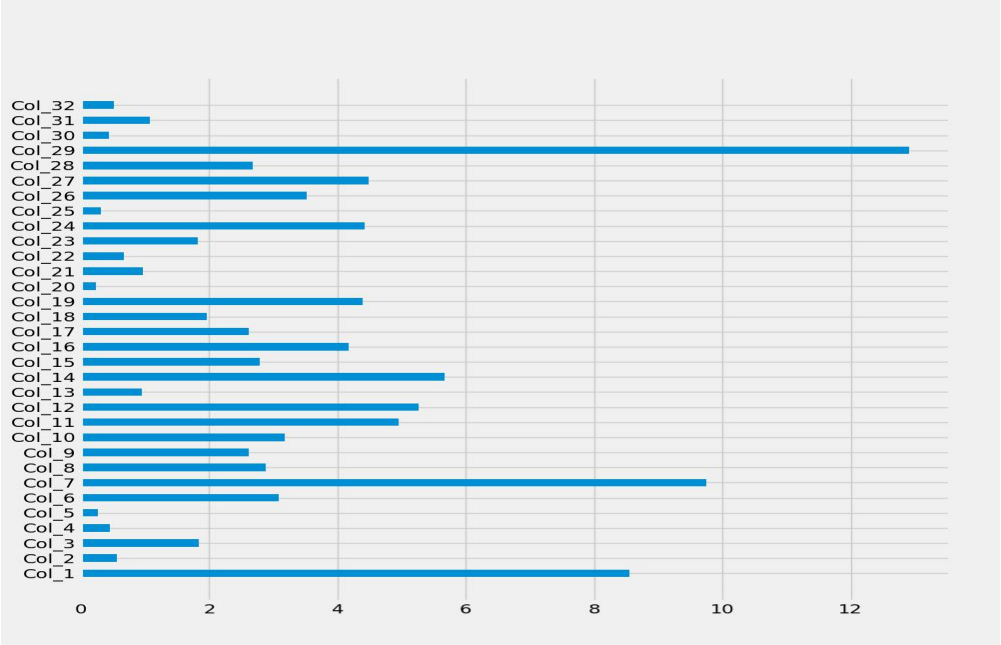


图 1 标签编码后特征的重要属性值

2.4 SVR 模型回归

SVR 同 SVC 一样，是从训练数据中选取一部分更加有效地支持向量，根据这些训练样本的值通过回归分析预测目标。分类和回归问题是有监督机器学习中最重要的两类任务。与分类的输出是有限个离散的值不同的是，回归模型的输出在一定范围内是连续的。

在 SVM 模型中边界上的点以及两条边界内部违反 margin 的点被当做支持向量，并且在后续的预测中起作用；在 SVR 模型中边界上的点以及两条边界以外的点被当做支持向量，在预测中起作用。按照对偶形式的表示，最终的模型是所有训练样本的线性组合，其他不是支持向量的点的权重为 0。

对于 SVR 的输入特征，分别采用独立热编码+PCA 产生的特征以均值编码后产生的特征。对于 SVR 的核函数分别采用线性核函数、多项式核函数以及径向基核函数进行非线性映射。

主要代码如下：

```
def sk_svm_train(self):
linear_svr = SVR(kernel='linear')  # 线性核函数初始化的 SVR
linear_svr.fit(self.training_x, self.training_y)
linear_svr_y_predict = linear_svr.predict(self.testing_x)

poly_svr = SVR(kernel='poly')  # 多项式核函数初始化的 SVR
poly_svr.fit(self.training_x, self.training_y)
poly_svr_y_predict = poly_svr.predict(self.testing_x)

rbf_svr = SVR(kernel='rbf', C=1e2, gamma=0.1)  # 径向基核函数初始化的 SVR
rbf_svr.fit(self.training_x, self.training_y)
rbf_svr_y_predict = rbf_svr.predict(self.testing_x)

print('R-squared value of linear SVR is', linear_svr.score(self.training_x, self.training_y))
print(' ')
print('R-squared value of Poly SVR is', poly_svr.score(self.training_x, self.training_y))
print(' ')
print('R-squared value of RBF SVR is', rbf_svr.score(self.training_x, self.training_y))

return linear_svr_y_predict, poly_svr_y_predict, rbf_svr_y_predict
```

运用 r^2 来对最终得到的模型进行评分，其中越接近 1 代表模型拟合的实际效果越好。独立热编码+PCA 特征的模型结果如图 2 所示，其中 PCA 保留了原来特征的 80%。均值编码后产生的特征的模型结果如图 3 所示。

```
R-squared value of linear SVR is -0.028992923296717654

R-squared value of Poly SVR is -0.04300022745983423

R-squared value of RBF SVR is 0.2738776288759984
```

图 2 独立热编码+PCA 特征的模型结果

```
R-squared value of linear SVR is -0.018302982099942522

R-squared value of Poly SVR is -0.07276239618093294

R-squared value of RBF SVR is 0.4894300034674336
```

图 3 均值编码特征的模型结果

可以发现，模型的拟合效果并不是很好，径向基为核函数的模型相对来说有一些参考意义，用该模型结合均值编码特征对测试数据进行预测，最终上传最终结果至 Lintcode，结果如图 4 所示，最终测试结果的 RMSE 为 3.96487。

图 4 SVR 模型预测结果

2.5 决策树模型回归

决策树是一种基本的分类与回归方法。回归决策树主要指CART(classification and regression tree)算法，内部结点特征的取值为“是”和“否”， 为二叉树结构。回归树是将特征空间划分成若干单元，每一个划分单元有一个特定的输出。因为每个结点都是“是”和“否”的判断，所以划分的边界是平行于坐标轴的。对于测试数据，我们只要按照特征将其归到某个单元，便得到对应的输出值。

算法原理：

在训练数据集所在的输入空间中，递归地将每个区域划分为两个子区域并决定每个子区域上的输出值，构建二叉决策树：

(1) 选择最优切分变量 j 与切分点 s，求解式（5）。

$$\min_{j,s}[\min_{c_1}\sum_{x_i\in R_1(j,s)}(y_i-c_1)^2+\min_{c_2}\sum_{x_i\in R_2(j,s)}(y_i-c_2)^2] \tag{5}$$

遍历变量 j，对固定的切分变量 j 扫描切分点 s，选择使上式达到最小值的对(j,s).

(2) 用选定的对(j,s)划分区域并决定相应的输出值：

$$\hat{c}_m=\frac{1}{N_m}\sum_{x_i\in R_m(j,s)}y_i, x\in R_m, m=1,2 \tag{6}$$

(3) 继续对两个子区域调用步骤(1),(2)，直至满足停止条件.

(4) 将输入空间划分为 M 个区域 R1,R2,...,RM，生成决策树：

$$f(x)=\sum_{m=1}^{M}\hat{c}_m I(x\in R_m), I=\begin{cases}1, if(x\in R_m)\\0, if(x\notin R_m)\end{cases} \tag{7}$$

主要代码如下：

```
    def AdaBoosting(self):
regr_1 = DecisionTreeRegressor(max_depth=4)
regr_2 = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                           n_estimators=50, random_state=self.rng)
regr_1.fit(self.training_x, self.training_y)
regr_2.fit(self.training_x, self.training_y)
y_1 = regr_1.predict(self.testing_x)
y_2 = regr_2.predict(self.testing_x)
for i in range(len(y_1)):
    y_1[i] = np.round(y_1[i])
for i in range(len(y_2)):
```

```
            y_2[i] = np.round(y_2[i])
        data_1 = y_1
        data_2 = y_2
        dataframe = pd.DataFrame({'Id': self.id, 'Score_1': data_1, 'Score_2': data_2})
        dataframe.to_csv("ada_result.csv", index=False, sep=',')
```

经过反复调试，决策数深度为 4 时，实际的回归效果最好。上传最终的预测结果至 Lintcode，得到的结果如图 5 所示,最终得到的测试结果 RMSE 为 3.93564。

图 5 决策树模型预测结果

2.6 神经网络回归

不难没发现由于原始数据的特征维数还是很高的，所以应用传统的回归模型得到的实际结果并不出众，于是考虑应用深度学习来解决实际的问题。对于求解实际的非线性问题来说，神经网络有着自己独有的优势。

对于选择神经网络为预测模型，有两个需要首要考虑的点，首先是确定网络结构，其次是选择具体的超参数。

基于 2.3 节中已经对于特征提取做了分析，针对均值编码选定了 20 个数据的主要特征，因此在网络上选择全联接神经网络，输入层为 20 维的特征向量。增加两个隐层，第一层 30 个节点，第二层 10 个节点，最终连接输出为 1 的输出层。

针对独立热编码，选择数据的全部 32 维特征，采用并行的结构。针对于未进行编码的 16 个数据，增加节点数为 20 以及 14 的隐藏层，连接输出为 1 的输出层。针对独立热编码后的 95 维特征，增加节点数为 100，50 以及 5 的隐藏层，连接输出为 1 的输出层。

2.6.1 超参数优化

针对神经网络的结构，采用自动化的方式搜索其超参数。设置的待定参数分别有 epochs，batch-size，optimizer 以及 activation。首先，确定 epochs，batch-size。

具体代码如下：

```
def epochs_and_batch(self):
    # fix random seed for reproducibility
    seed = 7
    numpy.random.seed(seed)
    # split into input (X) and output (Y) variables
    X = self.training_x
    Y = self.training_y
    # create model
    model = KerasRegressor(build_fn=self.create_model_bat, verbose=0)
```

```
        # define the grid search parameters
        batch_size = [100, 600, 800, 1000]
        epochs = [100, 500, 1000]
        param_grid = dict(batch_size=batch_size, epochs=epochs)
        grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=1)
        grid_result = grid.fit(X, Y)
        # summarize results
        print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
        means = grid_result.cv_results_['mean_test_score']
        stds = grid_result.cv_results_['std_test_score']
        params = grid_result.cv_results_['params']
        for mean, stdev, param in zip(means, stds, params):
            print("%f (%f) with: %r" % (mean, stdev, param))
```

最终得到两种编码形式下均有 epochs=100，batch-size 取 800 时，取得最小的 RMSE。

自动化调参确定 optimizer。具体代码如下：

```
    def optimizer(self):
        # fix random seed for reproducibility
        seed = 7
        numpy.random.seed(seed)
        # split into input (X) and output (Y) variables
        X = self.training_x
        Y = self.training_y
        # create model
        model = KerasRegressor(build_fn=self.create_model_op, epochs=100, batch_size=800,
verbose=0)
        # define the grid search parameters
        optimizer = ['SGD', 'RMSprop', 'Adagrad', 'Adadelta', 'Adam', 'Adamax', 'Nadam']
        param_grid = dict(optimizer=optimizer)
        grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=1)
        grid_result = grid.fit(X, Y)
        # summarize results
        print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
        means = grid_result.cv_results_['mean_test_score']
        stds = grid_result.cv_results_['std_test_score']
        params = grid_result.cv_results_['params']
        for mean, stdev, param in zip(means, stds, params):
            print("%f (%f) with: %r" % (mean, stdev, param))
```

最终得到两种编码形式下均有 optimizer=adam 时，取得最小的 RMSE。

自动化调参确定 activation。具体代码如下：

```
        def activation(self):
        # fix random seed for reproducibility
        seed = 7
```

```
        numpy.random.seed(seed)
        # load dataset
        # split into input (X) and output (Y) variables
        X = self.training_x
        Y = self.training_y
        # create model
        model = KerasRegressor(build_fn=self.create_model_act, epochs=100, batch_size=800,
verbose=0)
        # define the grid search parameters
        activation = ['softmax', 'softplus', 'softsign', 'relu', 'tanh', 'sigmoid', 'hard_sigmoid', 'linear',
'selu']
        param_grid = dict(activation=activation)
        grid = GridSearchCV(estimator=model, param_grid=param_grid, n_jobs=1)
        grid_result = grid.fit(X, Y)
        # summarize results
        print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_))
        means = grid_result.cv_results_['mean_test_score']
        stds = grid_result.cv_results_['std_test_score']
        params = grid_result.cv_results_['params']
        for mean, stdev, param in zip(means, stds, params):
            print("%f (%f) with: %r" % (mean, stdev, param))
```

最终得到均值编码形式下有 activation=selu 时，取得最小的 RMSE。独立热编码形式下有 activation=softplus，取得最小的 RMSE。

因此根据以上获得参数，在 keras 架构下建立神经网络模型并进行最终预测。

2.6.2 均值编码神经网络预测

均值编码形式神经网络建立的具体代码如下：

```
    def mlp_train_mean(self):
        model = Sequential()
        model.add(Dense(10,input_shape=(20,)))
        model.add(Dense(5))
        model.add(Dense(1, activation='selu'))
        model.add(Dropout(0.1))
        # Compile model
        model.compile(loss='mse', optimizer='adam', metrics=['mae'])
        tensorboard = TensorBoard(log_dir='mean_log')
        checkpoint = ModelCheckpoint(filepath=MEAN_MODEL_PATH, monitor='val_loss',
mode='auto')
        callback_lists = [tensorboard, checkpoint]

        history = model.fit(self.training_x, self.training_y, verbose=2,
                                        epochs=100, batch_size=800,
                                        validation_split=0.2,
                                        callbacks=callback_lists)
```

最终得到的训练和验证 MSE 值分别为 15.366 和 15.479。上传最终的预测结果至 Lintcode，得到的结果如图 6 所示,最终得到的测试结果 RMSE 为 3.82245。

图 6 均值编码神经网络预测结果

2.6.3 独立热编码神经网络预测

独立热编码形式神经网络建立的具体代码如下：

```
    def mlp_train(self):
        input_1 = Input(shape=(self.training_x_1.shape[1],), name='f_in')
        x_1 = Dense(output_dim=20)(input_1)
        x_2 = Dense(output_dim=10)(x_1)
        x_out = Dense(output_dim=1, activation='softplus')(x_2)
        x_out = Dropout(0.1)(x_out)
        input_2 = Input(shape=(self.training_x_2.shape[1],), name='r_in')
        y_1 = Dense(output_dim=100)(input_2)
        y_2 = Dense(output_dim=60)(y_1)
        y_3 = Dense(output_dim=5)(y_2)
        y_out = Dense(output_dim=1, activation='softplus')(y_3)
        y_out = Dropout(0.1)(y_out)
        concatenated = keras.layers.concatenate([x_out, y_out])
        out = Dense(output_dim=1, activation='selu')(concatenated)
        merged_model = Model(inputs=[input_1, input_2], outputs=[out])
        merged_model.compile(loss='mse', optimizer='adam', metrics=['mae'])
        tensorboard = TensorBoard(log_dir='mlp_log')
        checkpoint = ModelCheckpoint(filepath=MLP_MODEL_PATH, monitor='val_loss',
mode='auto')
        callback_lists = [tensorboard, checkpoint]

        history = merged_model.fit({'f_in': self.training_x_1, 'r_in': self.training_x_2},
self.training_y, verbose=2, epochs=1000, batch_size=800,
                                   validation_split=0.25,
                                   callbacks=callback_lists)
```

最终得到的训练和验证 MSE 值分别为 14.472 和 14.876。上传最终的预测结果至 Lintcode，得到的结果如图 7 所示,最终得到的测试结果 RMSE 为 3.80779。

图 7 独立热编码神经网络预测结果

3 小组分工

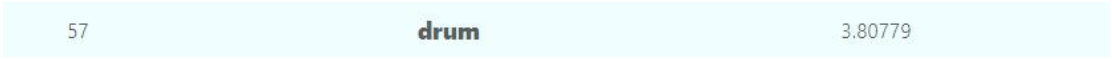
程序设计及编写：庄屹
程序调试：庄屹
作业报告：庄屹

4 作业总结

这次作业通过运用不同方法对于一个回归问题进行了实践预测。比较了传统的回归模型方法以及深度学习的方法。理解了 SVR，决策树以及神经网络在解决实际问题中如何运用。实验后发现对于特征空间较大的问题来说，神经网络模型略有优势。

当然本次实验还有很多不足，比如在特征的选取的问题上，并没有进行太深入的研究比较，编码方式的运用也未必是最佳的。没有考虑到特征于特征之间的关系以及单一和组合特征于实际结果之间的关系。传统模型中的参数优化以及神经网络的结构确定也缺乏一些实验性的论证。

总体来说，这次的作业加深了我对于回归模型更清晰的认识，也对于模式识别课程中所学到的相关原理和计算方法有了进一步实践上的体会。但因为能力有限，最终的结果在 Lintcode 不算很理想，有非常大的优化空间。



附：文件说明

本次作业共包括：

1 大作业报告

2 数据集以及程序代码，具体如下：

- data_processing 数据预处理及切割
  - correlation_test.py 检验特征重要性
  - data_process.py 数据预处理
  - split_dataset 训练集和测试集划分
- dataset 原始以及处理过后的数据集
- method 模型以及方法
  - adaboost.py 决策树模型
  - svr.py SVR 模型
  - mlp.py 神经网络模型
  - pca.py 主成分分析算法
- parameters_adjusting 参数调整
  - optimize_parameters.py 神经网络超参数优化
- util.py 路径