通过 Python 实现 BP 神经网络进行北京房价预测

摘要：本文利用 1999-2015 年北京市历史房价数据数据和同一时间段内该地区人口等其他影响因素数据，研究分析了各影响因素对北京市房价预测准确度的影响。在 BP 神经网络模型基础上，本文提出了一种利用前馈神经网络以及 BP 算法对房价与各种因素关系的预测模型通过对比和优化，我们选择了平均表现效果最好的模型结构及参数，对后一年（精确到每周）北京市房价走势进行了预测

关键词：前馈神经网络，反向传播算法，房价预测，深度学习

1. 目录

2. 问题重述

2.1. 问题背景

2001 年起的奥运会的成功申请与举办、2014 年 APEC 会议的召开，首都北京房地产在刚性需求巨大的同时，面临着房屋居住条件、社区基础设施建设进一步改善的巨大挑战。近几年，由于更多外来人口进京，而北京市土地资源也趋于饱和，北京市房价走势的变化引发了人们空前的关注，也都曾被或主观或客观地分析预测过。数据挖掘与人工神经网络的问世与应用确保了预言的准确性与科学性。因此对于预估，数据挖掘将会是一个可行的方法。

3. 数据来源及模型目标

3.1. 数据来源与预处理

本研究旨在基于相关数据对我们所在城市北京的房价走势作出预测，我们通过网络搜集获得了一些数据：

通过网络爬虫（见附件“crawler_of_price.py”）获取了各省市关于房价的数据。

通过国家统计局和北京市数据局获得了 1999 年-2015 年的北京市人口、全国 GDP、北京市人均收入、北京市房地产开发投资额等数据。

3.1.1. 近期房价数据与使用

在这近二十年间，北京的房价经历着之前人们无法想象的变化，虽然北京房价整体呈上涨趋势，但是其间不乏一些出其不意的波动和下跌，因而也是我们最为关注的。因为 2022 年冬奥会在即，我们认为之前几年的房价数据也非常有参考意义。因此，我们访问到了国家统计局，选取了自 1999 年到 2015 年北京市的房价信息，希望通过神经网络进行训练，得到房价关于时间的二维图像，进而对北京房价的未来走势进行进一步预测。

3.1.2. 其他影响因素的选取与预处理

一个地区的房价与地区经济状况和房地产的开发情况密切相关。

我们选取了与房价金额存在潜在关联的参数（见表）作为候选，并访问了北京市统计局和国家数据局，并通过爬虫工具搜寻了一些第三方网站，以获取相关影响因素的有效数据。但是，部分数据有残缺、遗漏的现象，且部分参数种类明显不符合北京市的情况。结合以上选取标准和社会学视角，我们缩小了范围，最终确定了除大赛所提示的流入流出人口以外我们认为最有必要考虑到的参数：GDP、在岗职工人均收入、房地产开发投资额。

其中，年末总人口数据来自国家统计局，GDP、常住人口数量、在岗工人人均收入、房地产开发投资额的数据来自北京市统计局。以上几个参数我们所选取的数据皆与 2.1.1 节中所提北京房价数据跨度一致（1999 年—2015 年）。

3.1.3. 原始数据的归一化处理

为了适应神经元激活函数 sigmoid 函数，我们对数据进行了[0,1]之间的归一化：

$$\bar{x}_i = \frac{x_i - x_{min}}{x_{max} - x_{min}}$$

3.2. 模型预测值评估

我们主要利用了 BP 算法进行模型评估，详见 3.3.2 节。

4. 神经网络模型

4.1. 神经网络简介

人工神经网络(Artificial Neural Network，ANN)，简称神经网络，是一种模仿生物神经网络的结构和功能的运算模型，可通过建立某种简单的模型，用于对表达式未知的复杂函数进行估计和近似。人工神经网络中称作神经元，连接在一起形成一个类似生物神经网络的网状结构。前馈神经网络 (Feedforward Neural Network, FNN)是最早发明、最简约的人工神经网络类型，也是本实验所采取的模型。前馈神经网络的结构通常包含一个输入层，一个或多个隐含层，和一个输出层，每个神经元有独立的权重和偏置。 其中，隐藏层便是特征空间，隐藏层节点的个数就是特征空间的维数，它连接了输入和输出层，连接的权重表示了前一层对后一层变量大小的贡献程度。数据经过计算，最终传入输出层。 BP 神经网络是一种按误差反向传播训练的前馈网络，其算法称为 BP 算法（3.节具体说明了 BP 算法在本研究中的具体使用），它的基本思想是梯度下降法，利用梯度搜索技术，以期使网络的实际输出值和期望输出值的均方误差为最小。在训练过程中，输入层将数据传入神经网络，网络通过训练不断地调整权重和偏置， 以不断降低输出层的数据与预期数据之间的误差为目的，最终得到最优的权重大小，得以后续使用。

4.2. 针对本问题的神经网络模型思路概述

对于本问题，北京市的房价受到多方面因素的影响，与经济发展、人口等密切相关。建立神经网络模型并不断调整神经元间连接的权重及偏置项大小进行反复训练，最终目标是使人工神经网络在我们输入所需时间的相关因素数据后，能够得到我们期望预测的该时间北京的房价。因此我们引入了反向传播 (back propagation)算法来解决这个问题。

4.3. 模型建立

4.3.1. 模型基础结构调试

BP 神经网络结构调整包括隐含层层数和个隐含层神经元数的调试。由于我们所用数据样本量较少，推导函数关系也相对容易，对于前者，我们依据最简化原则选取一层为隐含层层数。对于后者，我们以 5 和 50 两个不同的神经元数分别做了实验，比如在探究房价与时间关系时，我们分别使用全部数据中的 70%的数据对未来的房价走势进行训练，并对其余 30%进行预测，并将结果与其年份的真实数据相对比，以验证预测的准确性。我们发现，两者预测精度差别不大，且对于 50 个神经元，较少的数据更容易出现过拟合的现象，损失模型的测试集精度。结合我们所掌握的数据情况，我们最终将隐含层神经元数定为 5。

4.3.2. BP 反向传播算法

反向传播算法在本模型中被用于模型的优化，其中包括正向传播与反向传播。首先我们随机初始化连接权重，对某一训练样本进行一次前馈过程得到各神经元的输出。

前向传播

1.输入层->隐含层

首先，一组输入$x_1, x_2, \ldots, x_m$来到输入层，然后通过与隐藏层的连接权重w产生一组数据$net_{h_1}, net_{h_2}, \ldots, net_{h_m}$，其中$s$作为隐藏层的输入，然后通过隐藏层节点的 sigmoind θ ()激活函数后变为$\text{out}_{h_j} = \theta\ (net_{h_j})=\frac{1}{1+e^{-net_{hj}}}$其中，$\text{out}_{h_j}$表示隐藏层的第 j 个节点产生的输出。

2.隐含层->输出层

与上方相似，隐藏层节点的输出$out_{h_1}, out_{h_2}, \ldots, \text{out}_{h_m}$来到输出层，通过连接权重及代入激活函数的运算产生输出层不同节点的输出。

反向传播

1.误差计算

在回归模型中，误差通常定义为输出值与预期值的均方误差，本研究也将此作为模型的评估标准。

其定义式为：

$$E_{total}=\frac{1}{n}\sum_{i=1}^{n}(x_i-\widehat{x_i})^2$$

在本情况中，我们使用的公式为：

$$E_{o_1}=\frac{1}{2}(\text{target}_{o_1}-\text{out}_{o_1})^2$$

2.隐含层->输出层权重更新

对于连接隐藏层节点h_j和输出层节点o_j的权重参数w_j，我们通过计算整体误差对w_j的偏导进行体现，再利用链式法则进行运算：

$$\frac{\partial E_{total}}{\partial w_j}=\frac{\partial E_{total}}{\partial out_{oj}}\times\frac{\partial out_{oj}}{\partial net_{oj}}\times\frac{\partial net_{oj}}{\partial w_j}$$

最后，我们得到更新的w_j权值即：

$$w_j^{+}=w_j-\eta\frac{\partial E_{total}}{\partial w_j}$$（其中η为学习率，在 0-1 之间）

经过 10s（约 5000 次）的训练，我们将误差逐渐缩小，使输出值不断逼近真实值，最终得出最适合本模型的权重参数。

5. 模型求解及预测结果展示

我们将所有数据分为了训练集和测试集。训练集与测试集所占所有数据的比例分别为 0.7 和 0.3，只有训练集参与模型训练的过程，测试集用于训练结果的评估。训练集中当年的影响因素数据和当年的房价数据分别作为神经网络的输入数据和预测目标。测试集评估该神经网络体是否能够泛化，从而适用于新的数据。为了防止过拟合，我们利用控制训练时间和训练次数的方法，将每次训练的时间控制在 10.0s，训练的次数控制在 10000000 次，并每次训练计算测试集的精度，完成训练后选取测试集误差最小的模型，进行实际的预测，以避免过度训练所带来的误差。训练后所得到的权重及偏置大小即为经过训练后我们所得到最优化的模型，也是我们选择用于泛化、推广预测未来数据的模型。

5.1. 利用历史房价金额预测房价

我们将年份作为自变量，房价金额作为因变量代入了模型，绘制房价金额关于时间的连续图像来体现历史金额与未来房价金额的关联性。（图 1）

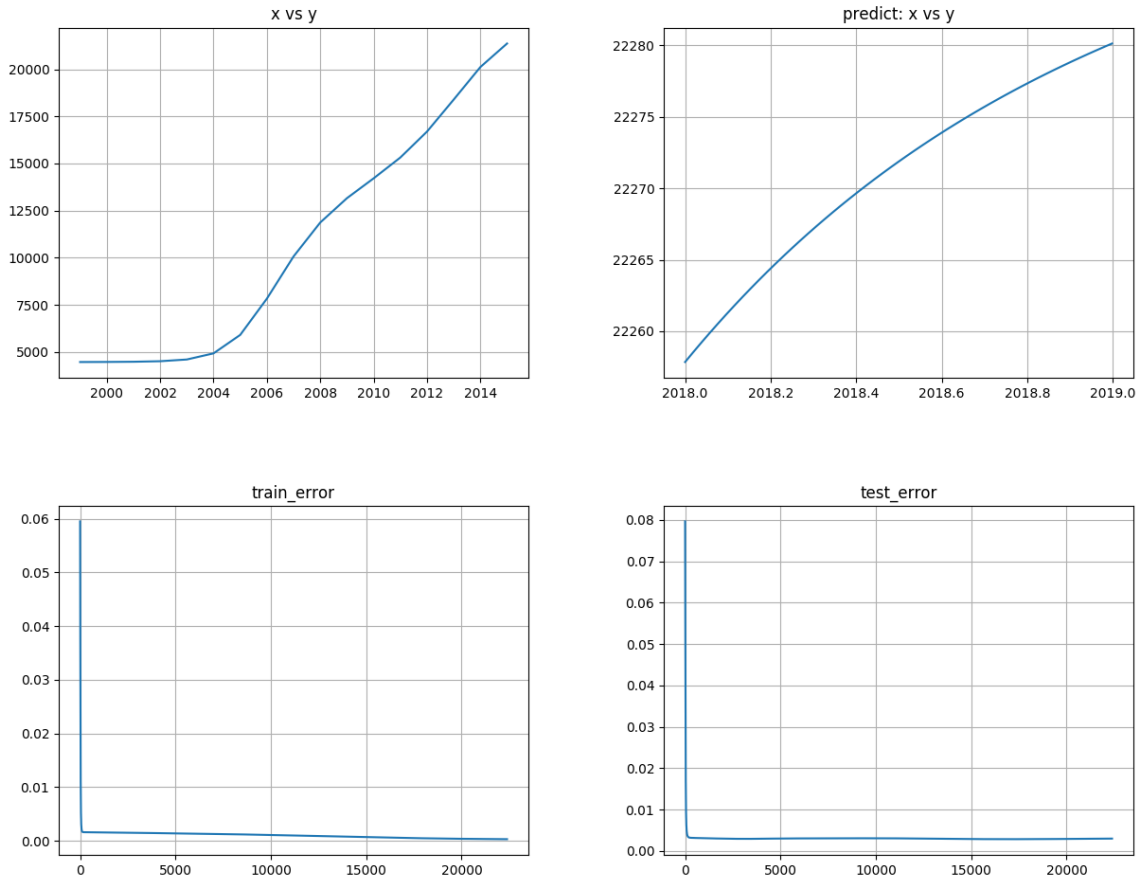

图 1 x 代表时间（年），y 代表房价（元）。上左为我们所绘制的房价数据关于时间的图像，上右为我们依据历史房价数据进行的 2018 全年的房价预测。下左、下右为我们模型训练过程中误差关于时间的变化图线，可见，经过训练本模型的预测精度不断在提升。

5.2. 基于北京流入流出人口数据的房价预测

我们首先利用神经网络绘制了模型通过历史数据学习预测了未来 1 年北京的流入流出人口数，并将其写入 Microsoft Excel 中以便后续使用（详见附件提交材料-最终数据-4-4.2-测试 1-predict）。

接着，我们将人口作为自变量，房价金额作为因变量代入了模型，绘制房价金额关于人口的连续图像来体现历史金额与未来房价金额的关联性。（图 2）

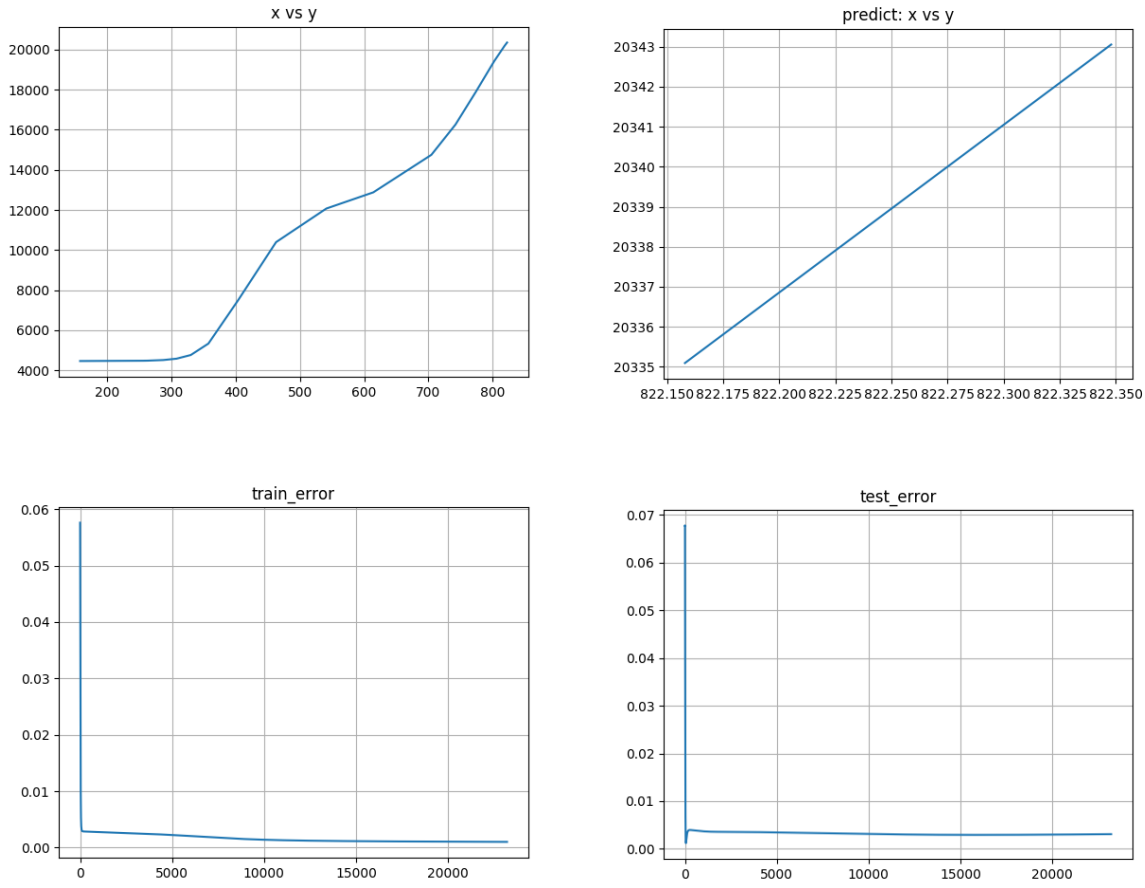

图 2 x 代表北京常住外来人口（百万），y 代表房价（元）。上左为我们所绘制的房价数据关于人口的图像，上右为我们依据历史房价数据及其与人口数的数学关系进行的房价预测。与图 1 相似，下左、下右为我们模型训练过程中误差关于时间的变化图线，可见，经过训练本模型的预测精度不断在提升。

5.3. 其他房价影响因素

参数的好处在于：相较于研究对象房价，它们较为真实、客观，因而可以被综合考虑以估计一个多因素影响的变量的情况。通过本文 2.1.2 节的讨论，我们选择了年末人口（第二问建议）、GDP、人均收入、房地产开发投资额三个因素进行研究。

对于未来的时间段，相关因素的数据也是未知的，因而我们在利用多因素预测房价之前，先用神经网络绘制了模型通过历史数据学习对未来 1 年 GDP、人均收入、房地产开发投资额的产生了预测，并将其写入 Microsoft Excel 中以便后续使用（详见附件提交材料-最终数据-4-4.2-测试 1-predict）。

5.4. 综合考虑其他因素的房价预测

通过 4.3 节其他因素数据的铺垫，我们将预测到的数据带入了含多神经节点（某房价待预测时间点其他四个因素的数据）的输入层和单输出节点（该时间待预测的房价）的输出层神经网络。

通过神经网络的训练和学习，我们绘制房价金额关于时间的连续图像最终体现了在考虑多因素的情况下本模型对北京近三月房价进行的预测。（图 3）

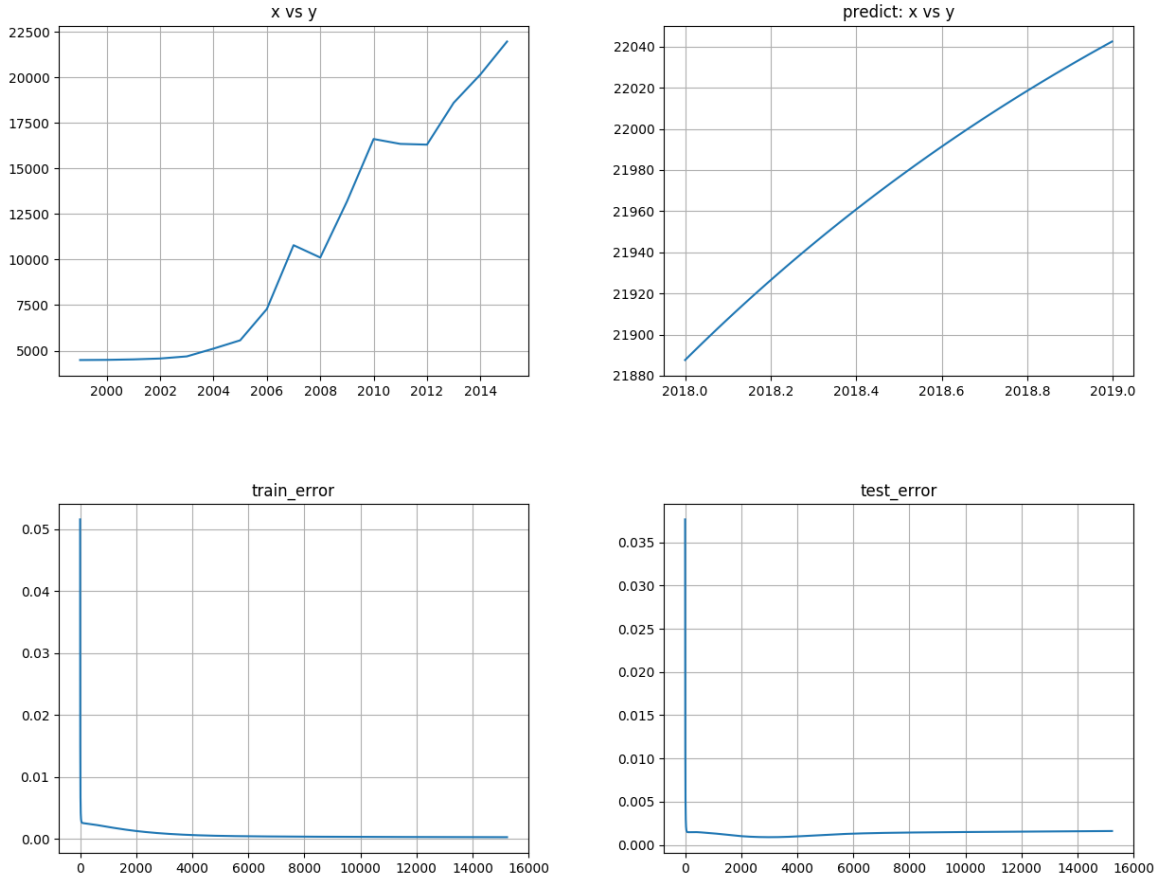

图 3 x 代表时间（年），y 代表房价（元）。上右为我们依据历史房价数据进行的 2018 全年的房价预测。下左、下右为我们模型训练过程中误差关于时间的变化图线，可见，经过训练本模型的预测精度不断在提升。

5.5. 考虑房价实用价值影响因素的量化

自 2008 年以来北京房价不断上升，越来越多的人开始“炒房”，导致住房价格畸形增长，与其实用价值完全不符。越来越多的年轻人成为“房奴”，住房成为一大问题。正如习主席所说，“房是用来住的，不是用来炒的”，房屋的价格应由其实用价值来决定。因此，如何量化影响房屋实用价值的因素是一个值得考虑的问题。在这里我们考虑交通便利对房屋实用价值的影响，并量化其对房屋价格做出的贡献。

我们通过进行相关性检验分析了各项影响因素对于房价影响的大小，下表展示了相关性检验的结果，其中的“死亡率”为对照，可以看出前四项分别对住宅商品房价格存在显著的相关性，而死亡率不存在相关性。

	常住外来人口数量	GDP	在岗职工人均收入	房地产开发投资额	死亡率

住宅商品房价格	0.973348	0.976512	0.973595	0.968299	-0.73725

6. 通过 Python 对于模型的实现

6.1. 代码解读

我们在开发程序时充分考虑到了代码的可读性，使用了丰富的中文注释。**如果读者阅读时间紧迫，可以只阅读“代码及注释”部分的注释**，该注释提供了结构清晰的程序实现方式的解读。

6.2. 程序架构

```
# 1 导入库
# 1.1 标准库
# 获取当前时间
# 数学
# 随机
# 矩阵
# 画图
# 读取 excel 的库
# 写入 excel 的库


# 2 全局变量
# 2.1 程序过程量
# 2.1.1 samples 集
# 2.1.2 归一化所需要的变量
# 2.1.3 其他
# 2.2 全局参数


# 3 数据的导入
# 3.1 读取 excel
    # 打开 xls 文件
    # 打开第一张表


# 3.2 把数据导入 samples[]
    # 分析有哪些 attributes
    # 读取第一行数据（表头）
    # 去掉第一个 label
    # 建立 temp_samples，并把 label 导入
    # 去除这一列的第一个元素（表头）
    # 对每项数据进行归一化处理

    # 从 data 里提取出 attributes，分别进行归一化处理，然后分别放到
temp_samples[].attributes[]里的每一项
    # 数据归一化处理
    # 对每项数据进行归一化处理（含 label）
    # 遍历每个 attributes
```

```
# 3.3 设置class Sample


# 3.4 将samples[]分成training_samples[]和testing_samples[]


# 4 神经网络
    # 初始化变量
    # 初始化
    # 输出预测值
    # 反向传播算法：调用预测函数，根据反向传播获取权重后前向预测，将结果与实际结果返回比较
误差
    # 计算测试集误差
    # 训练神经网络
    # 读取训练集误差最低时的神经网路体


# 5 数据的输出
# 5.1 画最终因变量关于自变量的图像


# 5.2 画测试集误差关于训练轮数的图像，展现泛化能力


# 5.3 画测试集误差关于训练轮数的图像，展现泛化能力# 画测试集误差关于训练轮数的图像，展现泛
化能力


# 5.4 写excel


# 5.5 预测并绘制图像
    # 将预测结果写入 excel
    # 将预测结果绘制图像


# 6 需要的函数
# 6.1 归一化


# 6.2 反归一化


# 6.3 随机函数


# 6.4 创建一个指定大小的矩阵


# 6.5 定义 sigmoid 函数


# 6.6 定义 sigmoid 的导数
```

```
# 7 主程序
    # 导入数据并进行数据处理
    # 产生随机数种子
    # 读取 Excel 第一张表（已知集的数据）
    # 导入 samples
    # 读取 Excel 第二张表（预测集的数据）
    # 导入 predict_samples
    # 把已知集划分为训练集和测试集
    # 建立神经网络并进行训练
    # 初始化神经网络：输入层，隐藏层，输出层元素个数
    # 可以更改
    # 恢复到训练最好时刻（还没有过拟合的时候）
    # 绘制输出的图像
    # 进行预测集的预测


# 8 程序起始执行
```

6.3. 代码及注释

```
# 1 导入库
# 1.1 标准库
import time  # 获取当前时间
import math  # 数学
import random  # 随机
import numpy as np  # 矩阵
import matplotlib.pyplot as plt  # 画图
import xlrd  # 读取 excel 的库
import xlwt  # 写入 excel 的库


# 2 全局变量
# 2.1 程序过程量
# 2.1.1 samples 集
samples = []
predict_samples = []
training_samples = []
testing_samples = []
# 2.1.2 归一化所需要的变量
norm_keys = []
norm_min_data = []
norm_max_data = []
# 2.1.3 其他
neural_network = None
# 2.2 全局参数
samples_split_rate = 0.7  # training_samples 占 samples 的比例 0-1 之间取值
folder_path = 'F:\近期事项文件\\2017.12.23 登峰杯数据挖掘\数据\最终数据\\1\\'
input_excel_path = '测试 1.xls'
output_excel_path = '输出 1.xls'
show_plot = False
training_cycle_limit = 10000000
training_time_limit = 10.0
```

```
# 3 数据的导入
# 3.1 读取 excel
def read_excel(path, sheet_num=0):
    data_sheet = xlrd.open_workbook(path)  # 打开 xls 文件
    first_sheet = data_sheet.sheets()[sheet_num]  # 打开第一张表
    return first_sheet


# 3.2 把数据导入 samples[]
def import_samples(data, what_samples='samples'):
    # 分析有哪些 attributes
    sheet_head = data.row(0)  # 读取第一行数据（表头）
    num_attributes = len(sheet_head) - 1  # 去掉第一个 label

    # 建立 temp_samples，并把 label 导入
    head = str(sheet_head[0])
    col = data.col_values(0)
    col.pop(0)  # 去除这一列的第一个元素（表头）
    normed_col = norm(col, head)  # 对每项数据进行归一化处理
    num_input_samples = len(normed_col)
    temp_samples = []
    for y in range(num_input_samples):
        temp_samples.append(Sample())
        temp_samples[y].label = normed_col[y]
        temp_samples[y].attributes = []

    # 从 data 里提取出 attributes，分别进行归一化处理，然后分别放到
temp_samples[].attributes[]里的每一项
    # 数据归一化处理
    # 对每项数据进行归一化处理（含 label）
    for x in range(1, num_attributes + 1):  # 遍历每个 attributes
        head = str(sheet_head[x])
        col = data.col_values(x)
        col.pop(0)
        normed_col = norm(col, head)
        for y in range(num_input_samples):
            temp_samples[y].attributes.append(normed_col[y])

    if what_samples == 'samples':
        global samples
        samples = temp_samples
    else:
        global predict_samples
        predict_samples = temp_samples


# 3.3 设置 class Sample
class Sample:
    attributes = None
    label = 0


# 3.4 将 samples[]分成 training_samples[]和 testing_samples[]
def split_samples():
    global training_samples, testing_samples
    temp_samples = []
```

```
    for sample in samples:
        temp_samples.append(sample)
    num_training_samples = int(len(samples) * samples_split_rate)
    for x in range(num_training_samples):
        index = int(random.random() * len(temp_samples))
        training_samples.append(temp_samples[index])
        temp_samples.pop(index)
    testing_samples = temp_samples


# 4 神经网络
class NeuralNetwork:
    def __init__(self):  # 初始化变量
        # 神经网络体变量
        self.input_n = 0
        self.hidden_n = 0
        self.output_n = 0
        self.input_cells = []
        self.hidden_cells = []
        self.output_cells = []
        self.input_weights = []
        self.output_weights = []
        self.input_correction = []
        self.output_correction = []
        # 最好的神经网络体变量
        self.best_test_error = 1
        self.best_cycle = 0
        self.best_input_weights = []
        self.best_output_weights = []
        # log
        self.log_training_error = []
        self.log_testing_error = []

    def setup(self, ni, nh, no):
        self.input_n = ni + 1  # 输入层+偏置项
        self.hidden_n = nh  # 隐含层
        self.output_n = no  # 输出层

        # 初始化神经元
        self.input_cells = [1.0] * self.input_n
        self.hidden_cells = [1.0] * self.hidden_n
        self.output_cells = [1.0] * self.output_n

        # 初始化连接边的边权
        self.input_weights = make_matrix(self.input_n, self.hidden_n)  # 邻接矩阵
存边权：输入层->隐藏层
        self.output_weights = make_matrix(self.hidden_n, self.output_n)  # 邻接矩
阵存边权：隐藏层->输出层

        # 随机初始化边权：为了反向传导做准备--->随机初始化的目的是使对称失效
        for i in range(self.input_n):
            for h in range(self.hidden_n):
                self.input_weights[i][h] = rand(-0.2, 0.2)  # 由输入层第 i 个元素到
隐藏层第 j 个元素的边权为随机值

        for h in range(self.hidden_n):
            for o in range(self.output_n):
```

```
                self.output_weights[h][o] = rand(-2.0, 2.0)  # 由隐藏层第i个元素到输出层第j个元素的边权为随机值
        # 保存校正矩阵，为了以后误差做调整
        self.input_correction = make_matrix(self.input_n, self.hidden_n)
        self.output_correction = make_matrix(self.hidden_n, self.output_n)

        # 输出预测值

    def predict(self, inputs):
        # 对输入层进行操作转化样本
        for i in range(self.input_n - 1):
            self.input_cells[i] = inputs[i]  # n个样本从0~n-1
        # 计算隐藏层的输出，每个节点最终的输出值就是权值*节点值的加权和
        for j in range(self.hidden_n):
            total = 0.0
            for i in range(self.input_n):
                total += self.input_cells[i] * self.input_weights[i][j]
                # 此处为何是先i再j，以隐含层节点做大循环，输入样本为小循环，是为了每一个隐藏节点计算一个输出值，传输到下一层
            self.hidden_cells[j] = sigmoid(total)  # 此节点的输出是前一层所有输入点和到该点之间的权值加权和

        for k in range(self.output_n):
            total = 0.0
            for j in range(self.hidden_n):
                total += self.hidden_cells[j] * self.output_weights[j][k]
            self.output_cells[k] = sigmoid(total)  # 获取输出层每个元素的值
        return self.output_cells[:]  # 最后输出层的结果返回

    # 反向传播算法：调用预测函数，根据反向传播获取权重后前向预测，将结果与实际结果返回比较误差
    def back_propagate(self, attributes, label, learn, correct):
        # 对输入样本做预测
        self.predict(attributes)  # 对实例进行预测
        output_deltas = [0.0] * self.output_n  # 初始化矩阵
        for o in range(self.output_n):
            error = label - self.output_cells[o]  # 正确结果和预测结果的误差：0,1，-1
            output_deltas[o] = sigmoid_derivate(self.output_cells[o]) * error  # 误差稳定在0~1内

        # 隐含层误差
        hidden_deltas = [0.0] * self.hidden_n
        for h in range(self.hidden_n):
            error = 0.0
            for o in range(self.output_n):
                error += output_deltas[o] * self.output_weights[h][o]
            hidden_deltas[h] = sigmoid_derivate(self.hidden_cells[h]) * error
            # 反向传播算法求W
        # 更新隐藏层->输出权重
        for h in range(self.hidden_n):
            for o in range(self.output_n):
                change = output_deltas[o] * self.hidden_cells[h]
                # 调整权重：上一层每个节点的权重学习*变化+矫正率
                self.output_weights[h][o] += learn * change + correct * self.output_correction[h][o]
```

```
                # 更新输入->隐藏层的权重
        for i in range(self.input_n):
            for h in range(self.hidden_n):
                change = hidden_deltas[h] * self.input_cells[i]
                self.input_weights[i][h] += learn * change + correct *
self.input_correction[i][h]
                self.input_correction[i][h] = change
                # 获取全局误差
        error = 0.5 * (label - self.output_cells[0]) ** 2  # 平方误差函数
        return error

    def test_error(self):
        error = 0.0
        for n in range(len(testing_samples)):
            error += 0.5 * (testing_samples[n].label -
self.predict(testing_samples[n].attributes)[0]) ** 2  # 平方误差函数
        error /= len(testing_samples)
        return error

    def train(self, nn_samples, limit, learn, correct, expire_time):
        start_time = time.time()
        self.log_training_error = []
        for i in range(int(limit)):
            error = 0.0
            for j in range(len(nn_samples)):  # 对输入层进行访问
                label = nn_samples[j].label
                attributes = nn_samples[j].attributes
                error += self.back_propagate(attributes, label, learn, correct)
# 样例，标签，学习率，正确阈值
            error /= len(nn_samples)
            self.log_training_error.append(error)
            test_error = self.test_error()
            self.log_testing_error.append(test_error)
            if test_error < self.best_test_error:
                self.best_test_error = test_error
                self.best_input_weights = self.input_weights
                self.best_output_weights = self.output_weights
                self.best_cycle = i

            if time.time() > start_time + expire_time:  # 超过秒数后结束训练过程
                print('cycles = %d; best_test_error = %f; best_cycle = %d' % (i,
self.best_test_error, self.best_cycle))
                break

    def load_best_parameters(self):
        self.input_weights = self.best_input_weights
        self.output_weights = self.best_output_weights


# 5 数据的输出
# 5.1 画最终因变量关于自变量的图像
def draw_result_plot():
    x = []
    y = []
    for n in range(len(samples)):
        raw_x = samples[n].attributes  # 画 label 关于 attributes 的第一项的函数
        y.append(de_norm(neural_network.predict(raw_x)[0], norm_keys[0]))
```

```
        x.append(de_norm(raw_x[0], norm_keys[1]))

    plt.plot(x, y)
    plt.grid(True)
    plt.title('x vs y')
    plt.savefig(folder_path + 'x vs y.png')
    if show_plot:
        plt.show()
    else:
        plt.close('all')


# 5.2 画测试集误差关于训练轮数的图像，展现泛化能力
def draw_test_error_plot():
    x = range(len(neural_network.log_testing_error))
    y = neural_network.log_testing_error

    plt.plot(x, y)
    plt.grid(True)
    plt.title('test_error')
    plt.savefig(folder_path + 'test_error.png')
    if show_plot:
        plt.show()
    else:
        plt.close('all')


# 5.3 画测试集误差关于训练轮数的图像，展现泛化能力# 画测试集误差关于训练轮数的图像，展现泛化能力
def draw_train_error_plot():
    x = range(len(neural_network.log_training_error))
    y = neural_network.log_training_error

    plt.plot(x, y)
    plt.grid(True)
    plt.title('train_error')
    plt.savefig(folder_path + 'train_error.png')
    if show_plot:
        plt.show()
    else:
        plt.close('all')


# 5.4 写excel
def write_excel(path, value):
    wb = xlwt.Workbook()
    sheet = wb.add_sheet("2003 测试表")
    for i in range(0, len(value)):
        for j in range(0, len(value[i])):
            sheet.write(i, j, value[i][j])
    wb.save(path)
    print("写入数据成功！")


# 5.5 预测并绘制图像
def predict_and_draw():
    x = []
    y = []
```

```
    excel_data = [[norm_keys[0], norm_keys[1]]]
    for n in range(len(predict_samples)):
        raw_x = predict_samples[n].attributes
        y.append(de_norm(neural_network.predict(raw_x)[0], norm_keys[0]))
        x.append(de_norm(raw_x[0], norm_keys[1]))
        excel_data.append([y[n], x[n]])
    # 将预测结果写入 excel
    write_excel(folder_path + output_excel_path, excel_data)
    # 将预测结果绘制图像
    plt.plot(x, y)
    plt.grid(True)
    plt.title('predict: x vs y')
    plt.savefig(folder_path + 'predict x vs y.png')
    if show_plot:
        plt.show()
    else:
        plt.close('all')
    print('The end')


# 6 需要的函数
# 6.1 归一化
def norm(data, key):
    global norm_keys
    if key in norm_keys:
        lock_index = norm_keys.index(key)
    else:
        norm_keys.append(key)
        lock_index = len(norm_keys) - 1  # 一把钥匙对应一把锁，但是需要知道锁的编号
    norm_max_data.append(max(data))
    norm_min_data.append(min(data))
    for n in range(len(data)):
        data[n] = (data[n] - norm_min_data[lock_index]) /
(norm_max_data[lock_index] - norm_min_data[lock_index])
    return data


# 6.2 反归一化
def de_norm(data, key):
    lock_index = norm_keys.index(key)
    data = data * (norm_max_data[lock_index] - norm_min_data[lock_index]) +
norm_min_data[lock_index]
    return data


# 6.3 随机函数
def rand(a, b):
    return (b - a) * random.random() + a


# 6.4 创建一个指定大小的矩阵
def make_matrix(m, n, fill=0.0):
    mat = []
    for i in range(m):
        mat.append([fill] * n)
    return mat
```

```
# 6.5 定义 sigmoid 函数
def sigmoid(x):
    return 1.0 / (1.0 + math.exp(-x))


# 6.6 定义 sigmoid 的导数
def sigmoid_derivate(x):
    return x * (1 - x)


# 7 主程序
def main():
    # 导入数据并进行数据处理
    random.seed(time.time())  # 产生随机数种子
    print('Importing')
    data = read_excel(folder_path + input_excel_path)  # 读取 Excel 第一张表（已知集的数据）
    import_samples(data)  # 导入 samples
    data = read_excel(folder_path + input_excel_path, 1)  # 读取 Excel 第二张表（预测集的数据）
    import_samples(data, 'predict_samples')  # 导入 predict_samples
    split_samples()  # 把已知集划分为训练集和测试集

    # 建立神经网络并进行训练
    global neural_network
    neural_network = NeuralNetwork()
    neural_network.setup(len(samples[0].attributes), 5, 1)  # 初始化神经网络：输入层，隐藏层，输出层元素个数
    print('Start training')
    neural_network.train(training_samples, training_cycle_limit, 1.0, 0.1, training_time_limit)  # 可以更改
    neural_network.load_best_parameters()  # 恢复到训练最好时刻（还没有过拟合的时候）

    # 绘制输出的图像
    draw_result_plot()
    draw_test_error_plot()
    draw_train_error_plot()

    # 进行预测集的预测
    predict_and_draw()


# 8 程序起始执行
if __name__ == '__main__':
    main()
```

7. 总结与展望

一个地区的房价金额，不仅是单纯经济方面的物价水平体现，也是人民生活水平、政治历史的展现。对于本次房价预测问题，我们将研究对象聚焦于我们所生活的城市北京。我们收集了 1999-2015 年共 17 年的北京房地产数据，以及这期间北京人口、GDP、人均收入等影响因素的数据，利用 BP 神经网络模型进行了今后房价走势的预测。我们发现 因素对于房价金额的预测有显著的正向作用。此外，本模型房价预测结果均以连续图像呈现，这样既可以了解房价整体走势，又可

以根据需要选择特定年份查看预见结果，这突出了本研究相较于众多主流媒体社会学角度的预测的科学性。

然而，由于时间关系，本文并没有讨论地价、地方财政预算、社会消费品零售总额、货币流通量等对房价金额可能存在重要影响的变量。而且，本文没有依据不同的自、因变量进行遍历调试，而是采用最简单的经验值来进行的网络结构设计，可能没有寻找到最优的模型。此外，本文所用于房价预测的数据有部份也是靠此模型预测得来的，因此可能产生潜在的误差。

我们的研究仍将继续。对于以上其他因素选取的不足，我们今后仍会对其加以讨论。对于模型的优化问题，我们决定在今后寻找和学习更优的模型（如 Elman 模型），对比本研究和未来研究的区别，不断去优化、完善我们的研究。

8. 参考文献

周志华，《机器学习》，清华大学出版社，9787302423287

http://www.bjstats.gov.cn/tjsj/，北京统计局 国家统计局北京调查总队

http://data.stats.gov.cn/，中华人民共和国国家统计局