Machine learning capstone project report: Hanoi house price prediction

Nguyen Trung Hieu, Nguyen Lan Cuong, Tran Duc Tri, Nguyen Duc Thanh, Cu Duy Hiep

Abstract

Predicting the price of real estate is a very intriguing problem that attracts many scientists, economists, politicians trying to tackle them because of its importance in securing financial security, controlling the economy, and guaranteeing the social security. These days, with the explosion of Artificial Intelligence (AI), especially in the field of Machine Learning (ML), many people try to apply ML algorithms into solving the real estate prediction problem. In this project, we try to predict the housing price in Hanoi using some of the ML methods including Random Forest, K-Nearest neighbor, Kernel Ridge regression, Gaussian Process, and our own proposed method which is called the ensemble network.

1. Introduction

In this project, we will use the Hanoi Housing Dataset 2020, which was a dataset crawled on the Internet about the real estate price in Hanoi. First, we will describe the dataset and its property, then how we preprocessed the data. After that, we present ML algorithms used in the project. Finally, the results of some experiments conducted to test models will be presented.

2. Data preprocessing and analysis

2.1. Data analysis

Hanoi Housing Dataset 2020 is a raw dataset containing 82.5 thousand records with 12 variables: Date, Address, District, Ward, House type, Legal status, Number of floors, Number of bedrooms, Area, Length, Width, Price per square meters.

We do some initial data preprocessing:

- Remove date, length, width column from the dataset because it has little impact on housing price and length, width column has lots of null value.
- Remove all value in the Area column that is either smaller than 30 or larger than 300 due to Hanoi regularization of housing area.
- Remove all value in the Number of floors that is larger than 8 also due to Hanoi regularization of number of floors.
- Only keep the inner districts (quận) in the district column and remove outer districts (huyện, thị xã). This is because we focus on predicting the house price of 12 inner districts, the outer districts house price is hugely different and only take 3% of the dataset.
- Remove all records that have null values.

After some initial preprocessing, we visualize our dataset with boxplot and countplot:

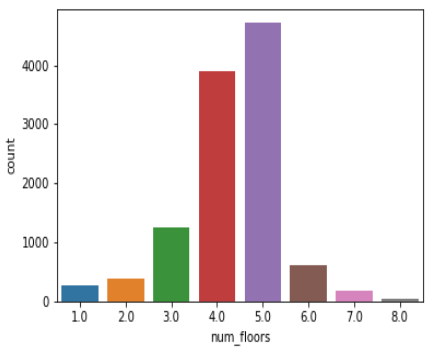

Fig 1: Number of floors count plot

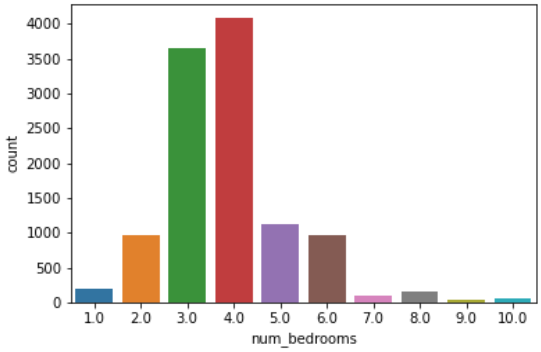

Fig 2: Number of bedrooms count plot

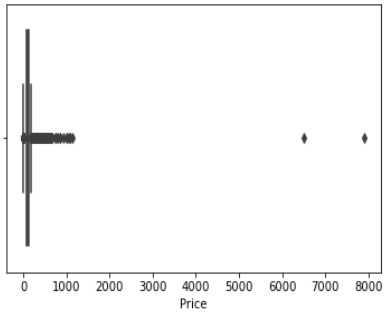

Fig 3: Boxplot of Price

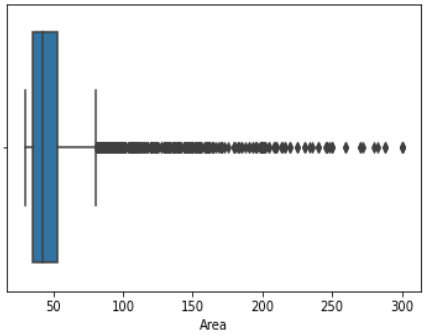

Fig 4: Boxplot of Area

The boxplot shows that Price variable have some extreme outliers, so we need to remove outliers of price. IQR outlier detection on price column helps to achieve this by removing the value x that:

$$x < Q_1 - 1.5 * IQR \; or \; x > Q_3 + 1.5 * IQR$$

where Q_1: 25^{th} percentiles, Q_3: 75^{th} percentiles, $IQR = Q_3 - Q_1$. After running IQR outlier detection and removal, the boxplot of Price:

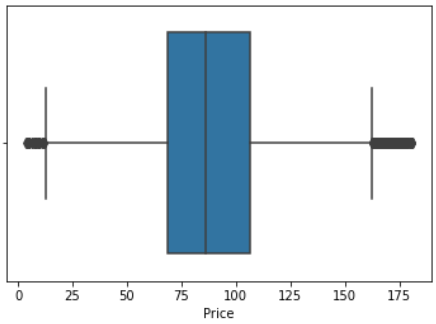

Fig 5: Boxplot on Price after running IQR

After preprocessing, we obtained 9,131 records.

With ward data, we use Binary encoding to process. First transform categorical value to ordinal value $0, 1, \dots, n$ with n is the number of unique wards. Then we transform the ordinal value to binary, which is a sequence of 1 and 0 lengths l. Split this sequence to l different columns, each column can be 0 or 1.

With house type data and legal status data, since this is ordinal data type, we will use ordinal encoding for these 2 columns:

- House types: Nhà ngõ, hẻm = 1, Nhà mặt phố, mặt tiền = 2, Nhà phố liền kề = 3, Nhà biệt thự = 4.
- Legal status: Không có sổ = 1, Giấy tờ khác = 2, Đang chờ sổ = 3, Đã có sổ = 4.

Since we use the ward data, we can for now drop the district column.

3. Machine learning approaches

We will discuss some machine learning approaches for the house price problem.

3.1. Random forest regressor

3.1.1.Decision tree regressor

Decision tree is a supervised machine learning algorithm where the mapping function is represented by a tree, at every node is an IF-ELSE rule. At each node an attribute d_i of the input is used to compare to threshold t_i, then the input is passed down the tree based on the comparison results on the threshold. At the leaf of the tree is the output of the prediction.

There are many algorithms to learn decision trees. In this project, we consider CART (Classification and Regression Trees) algorithm for learning. If we pass a training set $D = \{(X_n, y_n)\}_{n=1}^{N}$ pass to the tree, and at node i $D_i = \{(X_n, y_n) \in N_i\}$ is the data that reaches this node. We need to find a way to split this node to minimize the error on all the subtrees of this node.

If the feature is numeric, we can sort the unique values of feature j, then obtain vector $S_j = sorted\_unique(\{x_{ij}\})$ is the possible threshold for this node. For each possible threshold, we define the left and right split $D_i^L(j,t) = \{(X_n, y_n) \in N_i, x_{nj} \leq t\}$ and right split $D_i^R(j,t) = \{(X_n, y_n) \in N_i, x_{nj} > t\}$.

If the feature is categorical, we can split the attributes j based on unique categories of x_{ij}: $S_j = \{uniqe(\{x_{ij}\})\}$. Then we can define left and right split: $D_i^L(j,t) = \{(X_n, y_n) \in N_i, x_{nj}\ is\ t\}$, $D_i^R(j,t) = \{(X_n, y_n) \in N_i, x_{nj}\ is\ not\ t\}$.

Our goal is select an attribute at each node and a threshold to minimize the loss function:

$$H(j_i, t_i) = \frac{|D_i^L(j,t)|}{|D_i|} l\left(D_i^L(j,t)\right) + \frac{|D_i^R(j,t)|}{|D_i|} l\left(D_i^R(j,t)\right)$$

l is some function to measure the cost of node i. For our house price prediction problems, which is a regression problem, the appropriate loss function will be mean squared error:

$$l(D) = \frac{1}{|D|} \sum_{i \in D} (y_i - \bar{y})^2$$

with the prediction at each node is set as the average value $\bar{y} = \frac{1}{|D|} \sum_{i \in D} y_i$. This loss function can also be seen as minimizing the variance of the data after each split.

3.1.2. Random forest regressor

When training decision tree, if we grow the the tree deep enough, the tree can achieve 0 error on the training set, because we partition the search space into small enough regions where the outputs are constants. But those trees will usually perform poorly on unseen data, which is overfitting. To overcome this problem, we use Random forests regressor algorithm. Random forest is made of multiple decision trees, and the output of the forest will be averaged between these trees.

When learning random forests, we grow K decision trees with the following tactics:

- For each tree, we generate a dataset D_i to train by sampling with replacement from D.
- When we grow each individual tree, when choosing attributes and threshold to split a node, we only consider from a subset of attributes.
- We can limit the depth of each tree with a hyperparameter m.

After training random forest, we use the model for prediction by passing the input data to all *K* trees in the forest and average the results obtained from those trees to get the final output

3.2. Kernel Ridge regression:

First, we recall the method of ridge regression as follows:
Suppose that we have the set of inputs $\{(\mathbf{X}_i, y_i)\}$, where *i* denotes the index of the samples. The problem is to minimize:

$$\sum_{\boldsymbol{i}} (\boldsymbol{w^T X_i} - y_i)^2 + \lambda \boldsymbol{w}^{\mathrm{T}} \cdot \boldsymbol{w}$$

We take the derivative with respect to $\boldsymbol{w}$ of this loss function L, we have:

$$\frac{\partial L}{\partial \boldsymbol{w}} = \sum_{i} 2\ \mathbf{X_i}\left(\boldsymbol{w^T X_i} - y_i\right) + 2\lambda \boldsymbol{w}$$

Here, we will use gradient descent to find the optimal solution to the loss function. We will now show that we can express $\boldsymbol{w}$ **as a linear combination of all input vectors**.

$$\boldsymbol{w} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} \alpha_i\ \boldsymbol{X_i}$$

Since our loss function is convex, the gradient descent algorithm always guarantees the global optimal solution. So, here for convenience, we just pick our first iteration step to be the n-dimension 0-vector.

At each step we have the update $\boldsymbol{w_t} = \boldsymbol{w_{t-1}} - s\frac{\partial L}{\partial \boldsymbol{w}}$ (s is the iteration step).

The $\frac{\partial L}{\partial \boldsymbol{w}}$ term has two components. The first component is $\sum_i 2\ \boldsymbol{X_i}\left(\boldsymbol{w^T X_i} - y_i\right)$ is obviously a linear combination of input vectors. The second component has the term $\boldsymbol{w}$ which is also a linear combination of input vectors using induction hypothesis. So, we can prove by induction that the solution for this convex optimization problem is a linear combination of input vectors.
We write $\boldsymbol{w} = \mathbf{X}\alpha$ where α is a vector.

We also have the solution of ridge regression:

$$\boldsymbol{w} = (\boldsymbol{X X^T} + \lambda \mathbf{I})^{-\mathbf{1}} \boldsymbol{X y^T}$$

From two equations above, we can write $\mathbf{X}\alpha = (\boldsymbol{X X^T} + \lambda \mathbf{I})^{-\mathbf{1}} \boldsymbol{X y^T}$

We can use **Girard-Waring formula to expand** $(\boldsymbol{XX^T} + \lambda\mathbf{I})^{-\mathbf{1}}$ and after multiplying by $(\boldsymbol{X^TX})(\boldsymbol{X^TX})$ in both sides and simplify, we can obtain**:**

$$\alpha = (\boldsymbol{X^TX} + \beta^2\mathbf{I})^{-\mathbf{1}}\boldsymbol{y^T}$$

Now, we can let $\boldsymbol{X^TX} = \boldsymbol{K}$ and we can see that $\boldsymbol{K}$ now can be replaced by a kernel matrix (apply the kernel trick) because it is composed by inner product of all input vectors**.** In this project we consider some kernel matrix:

- Linear kernel: $\boldsymbol{K}(x_1, x_2) = x_1^T x_2$
- Laplacian kernel: $\boldsymbol{K}(x_1, x_2) = \exp\left(-\gamma\left||x_1 - x_2|\right|_1\right)$
- Polynomial kernel: $\boldsymbol{K}(x_1, x_2) = (\gamma x_1^T x_2 + 1)^3$
- Radial basis function kernel: $\boldsymbol{K}(x_1, x_2) = exp(-\gamma\left||x_1 - x_2|\right|_2^2)$

Thus, we have completed the process of kernelizing the ridge regression algorithms.

3.3. Gaussian Process

We can see the solution for ordinary least squares or (kernelized) ridge regression give us a predictive model for one particular parameter of $\boldsymbol{w}$**.**

In general, the posterior predictive distribution is given by the following formula:

$$P(Y\,|D, X) = \int_w P(Y, \boldsymbol{w}\,|D, X)\,\boldsymbol{dw} = \int_w P(Y|\boldsymbol{w}, D, X)\boldsymbol{P}(\boldsymbol{w}|D)\boldsymbol{dw}$$

Where Y is the predictive target, D is the dataset, X is the given attributes.

However, in general, this formula is intractable in closed form. But now, if we assume that we have a Gaussian prior and likelihood, we can obtain the mean and variance. In the ridge regression model, we use these assumptions and using Maximum a posteriori to estimate a point of $\boldsymbol{w}$. But now, we want to model the regression function directly, not just estimate a point by MAP as in ridge regression.

3.3.1. Gaussian Process definition

Definition: A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution.

A Gaussian process is completely specified by their second-order statistics (its mean and covariance function). We can define the mean function $m(\mathbf{x})$ and their covariance matrix $k(\mathbf{x}, \mathbf{x}')$ of a real process $f(\mathbf{x})$ as:

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})]$$
$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}) + m(\mathbf{x}))]$$

Without loss of generality, we usually take the mean function to be zero. This is possible by subtracting the sample mean. So, the gaussian process will be determined only by its covariance function.

The specification of the covariance function implies a distribution over functions. To see this, we can draw samples from the distribution of functions evaluated at any number of points. In detail, if we choose a number of input points $\boldsymbol{X}_*$ **we can generate a random Gaussian vector with covariance matrix**

$$\mathbf{f}_* \sim N\,(0, K(\boldsymbol{X}_*, \boldsymbol{X}_*)\,)$$

3.3.2 Gaussian process prediction

- **Prediction for noise-free observations**

We can write the joint distribution of training output $\mathbf{f}$, and testing output $\mathbf{f}_*$ as follows:

$$\begin{bmatrix}\mathbf{f}\\ \mathbf{f}_*\end{bmatrix} \sim N(0, \begin{bmatrix}K(\boldsymbol{X},\boldsymbol{X}) & K(\boldsymbol{X},\boldsymbol{X}_*)\\ K(\boldsymbol{X}_*,\boldsymbol{X}) & K(\boldsymbol{X}_*,\boldsymbol{X}_*)\end{bmatrix})$$

The distribution of $\mathbf{f}_*$ conditioning on the observations can be written as follows**:**

$$\mathbf{f}_*|\boldsymbol{X}_*,\boldsymbol{X},\mathbf{f} \sim N(K(\boldsymbol{X}_*,\boldsymbol{X})K(\boldsymbol{X},\boldsymbol{X})^{-1}\mathbf{f}, K(\boldsymbol{X}_*,\boldsymbol{X}_*) - K(\boldsymbol{X}_*,\boldsymbol{X})K(\boldsymbol{X},\boldsymbol{X})^{-1}K(\boldsymbol{X},\boldsymbol{X}_*))$$

- **Prediction of noisy observations**:

It is typical for a realistic model that we do not have access to the values themselves, but with the noisy version $y = f(\mathbf{x}) + \varepsilon$. Here we assume that the noise ε is also Gaussian distributed. So, we can write the covariance function $cov(y) = K(\boldsymbol{X},\boldsymbol{X}) + \sigma^2\mathbf{I}$. Here, the joint distribution between the target values and the function values can be written as

$$\begin{bmatrix}\mathbf{y}\\ \mathbf{f}_*\end{bmatrix} \sim N(0, \begin{bmatrix}K(\boldsymbol{X},\boldsymbol{X}) + \sigma^2\mathbf{I} & K(\boldsymbol{X},\boldsymbol{X}_*)\\ K(\boldsymbol{X}_*,\boldsymbol{X}) & K(\boldsymbol{X}_*,\boldsymbol{X}_*)\end{bmatrix})$$

Using the similar way as before to derive the function of $\mathbf{f}_*$ conditioning on the observations.
Now, with one test point $\mathbf{x}_*$, we write $\boldsymbol{k}(\mathbf{x}_*) = \boldsymbol{k}_*$ to denote the vector of covariances between this test point and the n training points. We derive the meaning and variance of $\mathbf{f}_*$

$$\mathbb{E}[\mathbf{f}_*] = \boldsymbol{k}_*^{\boldsymbol{T}}(\boldsymbol{K}(\boldsymbol{X},\boldsymbol{X}) + \sigma^2\mathbf{I})^{-1}\boldsymbol{y}$$

$$\mathbb{V}[\mathbf{f}_*] = \boldsymbol{k}(\mathbf{x}_*,\mathbf{x}_*) - \boldsymbol{k}_*^{\boldsymbol{T}}(\boldsymbol{K}(\boldsymbol{X},\boldsymbol{X}) + \sigma^2\mathbf{I})^{-1}\boldsymbol{k}_*$$

Another way to look at this equation of the mean is to see it as a linear combination of n kernel function as we see in kernelized ridge regression**.**

- **Cholesky factorization**

One more topic that I want to explain here is the Cholesky decomposition of a real positive-definite matrix. When **A** is a real symmetric positive-definite matrix (a property that a covariance matrix of a Gaussian distribution has), we can decompose it as $\mathbf{A} = \mathbf{L}\mathbf{L}^{\boldsymbol{T}}$**,** where **L** is a real lower triangular matrix with positive diagonal entries**.**

- **Algorithms for Gaussian process regression**

Here, we can dive into the Gaussian process regression algorithms, which is also the algorithm implemented by sklearn library-the library we use in this project.

Input: $\boldsymbol{X}$ (inputs), $\boldsymbol{y}$ (targets), $\boldsymbol{k}$ (covariance function), σ^2 (**noise level**), $\mathbf{x}_*$**(test input)**

1. $\boldsymbol{L} := \boldsymbol{cholesky}((\boldsymbol{K}(\boldsymbol{X},\boldsymbol{X}) + \sigma^2\mathbf{I}))$
2. $\boldsymbol{\alpha} := \boldsymbol{L}^{\boldsymbol{T}} \setminus (\boldsymbol{L} \setminus \boldsymbol{y})$
3. $\mathbb{E}[\mathbf{f}_*] := \boldsymbol{k}_*^{\boldsymbol{T}}\boldsymbol{\alpha}$
4. $\mathbf{v} := \boldsymbol{L} \setminus \boldsymbol{k}_*$
5. $\mathbb{V}[\mathbf{f}_*] := \boldsymbol{k}(\mathbf{x}_*,\mathbf{x}_*) - \mathbf{v}^{\boldsymbol{T}}\mathbf{v}$
6. $\log p(\boldsymbol{y}|\boldsymbol{X}) := -\frac{\boldsymbol{1}}{\boldsymbol{2}}\boldsymbol{y}^{\boldsymbol{T}}\boldsymbol{\alpha} - \sum_{\boldsymbol{i}}\log \boldsymbol{L}_{\boldsymbol{ij}} - \frac{\boldsymbol{n}}{\boldsymbol{2}}\log 2\pi$
7. **Return** $\mathbb{E}[\mathbf{f}_*]$(mean), $\mathbb{V}[\mathbf{f}_*]$ (variance), $\log p(\boldsymbol{y}|\boldsymbol{X})$ (Log marginal likelihood).

- **The marginal likelihood** is the integral of the likelihood times the prior

$$\log p(\boldsymbol{y}|\boldsymbol{X}) = \int p(\mathbf{y}|\mathbf{f}, X)p(\mathbf{f}|X)\, d\mathbf{f}$$

Under the Gaussian process, we have the prior is the Gaussian $\mathbf{f}|X \sim N(0, K)$ and the likelihood is the factorized Gaussian $\mathbf{y}|\mathbf{f} \sim \mathrm{N}(\mathbf{f}, \sigma^2\mathbf{I})$ so, we can write down the equation of computing the marginal likelihood as in the algorithm.

3.3.3 The covariance function – heart of Gaussian process

Because we can assume that the mean of Gaussian process is 0, so the Gaussian processes regression obtain their power through covariance function. There are three main ways to choose a good covariance function:

1. Expert knowledge (awesome to have, difficult to get)

2. Bayesian model selection (more possibly to face intractable integrals)
3. Cross-validation (time consuming but easy to implement)

We want to choose a covariance function so that **if the data is closer, their covariance matrix index is higher.** We can choose the covariance matrix as a kernel matrix because they have this property and between them, there are many other similar properties (positive semi-definite, symmetric…).

Here we will explain some simple kernels that we use in this project:

Definition: Stationary kernels — functions that depend only on the radial distance between points in some user-defined metric, and. Non-stationary kernels — functions that depend on the value of the input coordinates themselves.

- **RBF kernel**

  The RBF kernel is a stationary kernel. It is also known as the "squared exponential" kernel. It is parameterized by a length scale parameter $\boldsymbol{l} > 0$, which can either be a scalar (isotropic variant of the kernel) or a vector with the same number of dimensions as the inputs X (anisotropic variant of the kernel). The kernel is given by:

$$\boldsymbol{k(x_i, x_j) = exp(-\frac{d(x_i, x_j)^2}{2l^2})}$$

  where l is the length scale of the kernel and $d(\cdot,\cdot)$ is the Euclidean distance**.** This kernel is infinitely differentiable, which implies that GPs with this kernel as covariance function have mean square derivatives of all orders, and are thus very smooth

- **Dot product:**

  Dot product kernels**:** $\boldsymbol{k(x_i, x_j) = \sigma^2 + x_i \cdot x_j}$

  Sigma is set to be 1.0

- **White kernel:**

  This kernel is used to explain the noise in covariance matrix as an independent normal distribution. The formula for white kernel:

  White kernel**:** $\boldsymbol{k(x_i, x_j)} = \begin{cases} \boldsymbol{noiselevel}\ if\ x_i == x_j \\ \boldsymbol{0}\ otherwise \end{cases}$

  In this project, we will use noise level = 0.5

- **Matérn kernel**

  The Matérn kernel is a generalization of RBF kernel. It has an additional parameter υ which controls the smoothness of function. The smaller υ, the less smooth the approximated function is. As $\upsilon \to \infty$ the kernel become RBF kernel. The default value of υ is 1.5. The kernel is given by:

$$\boldsymbol{k(x_i, x_j) = \frac{1}{\Gamma(\upsilon)2^{\upsilon-1}}\left(\frac{\sqrt{2\upsilon}}{l} d(x_i, x_j)\right)^{\upsilon} K_{\upsilon}(\frac{\sqrt{2\upsilon}}{l} d(x_i, x_j))}$$

  Where $d(\cdot,\cdot)$ is the Euclidean distance, $\boldsymbol{K_{\upsilon}(\cdot)}$ is a modified Bessel function and $\boldsymbol{\Gamma(\cdot)}$ is the gamma function.

- **Rational Quadratic kernel**

  Rational Quadratic kernel can be seen as an infinite sum of RBF kernel with different characteristic length scales. It is parameterized by a length scale parameter $l > 0$ and a scale mixture parameter $\alpha > 0$ The kernel is given by

$$k(x_i, x_j) = (1 + \frac{d(x_i, x_j)^2}{2\alpha l^2})^{-\alpha}$$

where $d(\cdot,\cdot)$ is the Euclidean distance.

3.4. Ensemble neural network

In this section, we propose a method called ensemble network. The idea of this algorithm is very simple. We will explain the algorithm step-by-step:

1. Build an input and output layer of this neural network.
2. Build a set A of hidden layer with the same number of neurons and same number of neurons with the input layer.
3. Generate the set S containing all subsets of A.
4. With each subset S_i in S, build a neural network with the input layer, the hidden layer in S, and the output layer. Then train this neural network in training dataset.
5. Generate the full neural network as our final model, which combines the input layer, all hidden input layers in A, and the output layer. Test this model in the testing dataset.

Here, we use the loss function in the model is the mean-squared loss, the optimization algorithm is Adam optimization.

- Notes about Adam optimization

Motivation: When we want to train a neural network, we want to use a good optimization method. The most traditional optimization algorithm used is the stochastic gradient descent (SGD). One of the most challenging problems is to pick the value for the step size of SGD. If the step is too small, the convergence is slow and if it's too large, then we risk divergence or slow convergence due to oscillation. It's also true that, within a single neural network, we may well want to have different step sizes. As our networks become deep (with increasing numbers of layers) we can find that magnitude of the gradient of the loss with respect the weights in the last layer, may be substantially different from the gradient of the loss with respect to the weights in the first layer. Another problem is the exploding or vanishing gradients. in which the back-propagated gradient is much too big or small to be used in an update rule with the same step size.

So, we'll consider having an independent step-size parameter for each weight, and updating it based on a local view of how the gradient updates have been going.

Method: The Adam optimization algorithm is the combination of two ideas to solve our mentioned problem. The first one is the momentum, in which we try to "average" recent gradient updates, so that if they have been bouncing back and forth in some direction, we take out that component of the motion, and finally we can speed up the convergence. The second idea is a technique called Adadelta, in which we would like to take steps in parts of the space where the loss function is nearly flat (because there's no risk of taking too big a step due to the gradient being large and smaller steps when it is steep, and this idea is applied to each weight independently.

Here is the full algorithm of Adam (Adaptive momentum):

Require: α: Step size

Require: $\beta_1, \beta_2 \in [0,1)$: Exponential decay rates for the moment estimates

Require: $f(\theta)$: Stochastic objective function with parameters θ

Require: θ_0: Initial parameter vector

$m_0 \leftarrow 0$ (Initialize 1st moment vector)

$v_0 \leftarrow 0$ (Initialize 2nd moment vector)

$t \leftarrow 0$ (Initialize timestep)

while θ_t not converged do

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradient w.r.t stochastic objective at timestep t)

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot m_{t-1} + (1 - \beta_2) \cdot {g_t}^2$ (Update biased second raw moment estimate)

$\widehat{m_t} \leftarrow m_t / (1 - {\beta_1}^t)$ (Compute bias-corrected first moment estimate)

$\widehat{v_t} \leftarrow v_t / (1 - {\beta_2}^t)$(Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m_t} / (\sqrt{\widehat{m_t}} + \varepsilon)$(Update parameters)

end while

return θ_t (Resulting parameters)

(${g_t}^2$ indicates the element-wise square, ${\beta_1}^t$ and ${\beta_2}^t$ denote β_1 and β_2 to the power of t)

As suggested by the author a good default setting is $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 10^{-8}$.

Returning to our ensemble model, there are two hyperparameters that we need to choose for our model. The first one is the number of neurons in each layer, and the second one is the number of hidden layers that we need to generate. Using coordinate ascent method of optimization, we conclude that for this problem, the optimal number is 25 neurons in each layer and 3 hidden layers. We also tried to shuffle the order or take the subset of S in training phase, but these techniques did not improve the performance of the model. Here, we trained all the subset of S and taking the subset of S based on dictionary order.

By using this model, we observe significantly lower testing errors in our experiment compared to traditional neural networks. One explanation for this is that this method can be viewed as training an ensemble of neural network implicitly. Each layer in S is either used or not used, resulting in 2^L possible network. For each training step, one of these networks' weights will be updated. During testing, all networks are averaged.

One of the main disadvantages of this method is the time complexity. Because we need to train 2^L networks, the time need for training increases exponentially with the number of hidden layers, which provides a huge obstacle for increasing the complexity of the model.

4. Experiments

4.1. Experiments setup

For training and testing the model, we split the data into 2 parts: training data and testing data. The splitting ratio will be 8:2, so there are 7304 training samples and 1827 test samples.

In this project, we use Python version 3.7.13 as our programming language and multiple libraries for handling data and algorithm implementations:

- For handling numeric data, we use Numpy version 1.12.6
- For reading and preprocessing data comes in CSV files, we use Panda's version 1.3.5.
- For plotting and visualizing, we use Seaborn version 0.11.2
- For binary encoding, we use a special library categories encoder version 2.5.0
- For implementing machine learning method (Random Forest, Gaussian process, …) we use Scikit learn version 1.0.2
- For implementing ensemble neural networks, we use Tensorflow version 2.8.2

For the model assessment, we use 3 different metrics: root mean squared error (RMSE), mean absolute error (MAE), mean absolute percentage error (MAPE).

4.2. Hyperparameter tuning

In this section we will train all the models mentioned in section 3. For hyperparameter tuning, we will use 4 folds cross-validation on the training set. We use scikit-learn library for model implementation and cross validation.

4.2.1. Random forest regressor model

When training random forest regressor, we need to fine tune 2 hyperparameters: number of decision tree in the forest and maximum depth of each tree we grow. Perform grid search and cross validation we obtain:

	N = 50	N = 100	N = 150	N = 200	N = 250	N = 300	N = 350	**N = 400**
MAPE	0.277	0.276	0.276	0.276	0.276	0.276	0.276	**0.276**
RMSE	25.513	25.461	25.440	25.427	25.429	25.424	25.426	**25.422**
MAE	19.014	18.976	18.959	18.953	18.960	18.952	18.951	**18.949**

Table 1: Random Forest performance on different number of trees

	d = 5	d = 8	d = 11	d = 14	d = 17	**d = 20**	d = 23	d = 26	d = inf
MAPE	0.309	0.290	0.274	0.268	0.267	**0.267**	0.267	0.267	0.267
RMSE	27.530	26.530	25.280	24.904	24.864	**24.854**	24.856	24.865	24.865
MAE	21.003	19.846	18.874	18.846	18.389	**18.379**	18.380	18.385	18.384

Table 2: Random Forest performance on different maximum depth of each tree

Through grid search, we see that the change in number of trees has little impact on model prediction, while maximum depth has great impact. Varying number of trees from 50 to 400 but model performance remains almost the same. When we increase maximum depth, the performance improves quickly, but at depth = 20 model stop improving and remain almost the same after that. We find the optimal hyperparameter of the model is number of trees N = 400 and max depth of each tree in the forest is d = 20.

4.2.2. Kernel ridge regression

Here we test Kernel ridge regression with four different kernel functions: linear, Laplacian, RBF, Polynomial with the default value for the other two parameters: α – the coefficient of regularization (1.0) and gamma – the coefficient in defining the Laplacian, RBF and polynomial kernel function (0.1)

	Linear	RBF	Polynomial	**Laplacian**
MAPE	0.307	0.276	0.279	**0.268**
RMSE	28.12	26.48	26.38	**26.08**
MAE	21.39	19.48	19.61	**19.31**

Table 3: Kernel ridge performance on different kernels

After observing that Laplacian kernel provides the best result, we tune the parameter α coeffiecient using Laplacian kernel and receive the following result.

	0.001	0.025	**0.05**	0.1	0.25	0.5	1.0	2.0
MAPE	0.298	0.281	**0.281**	0.281	0.282	0.283	0.284	0.286
RMSE	29.13	26.59	**26.34**	26.35	26.35	26.35	26.42	26.55
MAE	20.93	19.73	**19.69**	19.70	19.73	19.82	19.91	20.02

Table 4: Kernel ridge performance on different α

We observe that gamma in the default value provides the best result for the Laplacian kernel and α= 0.35. Too large $\alpha = 2$ or too small $\alpha = 0.001$ all result in bad performance. When α is between 0.025 and 1, the model performance stays almost the same, the difference in the score is very small. Any α in this range might work as fine, but we can select the best α on RMSE and MAE results, which is $\alpha = 0.35$.

Note that in this experiment, we perform 4 – fold cross – validation on both tables but use different folds split, so the results in 2 tables are slightly different. If we use the same folds in two processes, our hyperparameter selection might be biased, so we split the fold differently each time.

4.2.3. Gaussian process:

In gaussian process we need to choose a suitable kernel for the problem. We perform cross – validation on the traning set with different kernels. We combine all these kernels with a WhiteNoise kernel has $noiselevel = 0.5$. The results is in the table below:

	Rational Quadratic	Dot Product	RBF	**Matérn**
MAPE	0.256	0.280	0.261	**0.255**
RMSE	25.96	27.65	26.31	**25.75**
MAE	19.48	21.17	19.94	**19.29**

Table 5: Gaussian process performance on different kernels

The Matérn kernel and Rational Quadratic kernel performance is the best, while RBF and Dot Product kernel has worse performance. Especially the Dot Product kernel, since this kernel is very simple (compute inner product), it might not capture the nature of the data. According to experimental results, we select Matérn kernel for the model.

4.2.4. Ensemble neural network

Through experiments with the ensemble neural network model, we found that the number of optimal layers int this project is 3, any larger number of layers will result in overfitting. With 3 layers, we test our model performance on different number of neurons with cross - validation:

	n = 5	n = 10	n = 15	n = 20	**n = 25**	n = 30
MAPE	0.260	0.245	0.241	0.233	**0.231**	0.236
RMSE	27.52	26.29	26.39	25.44	**25.27**	25.87
MAE	20.84	19.66	19.61	18.77	**18.53**	19.10

Table 6: Ensemble neural network on different number of neurons

The model with 25 neurons in each layer yields the best performance in terms of MAPE, MAE and RMSE. We will use this parameter in our final model.

4.3. Model training and comparison:

We train the model with the hyperparameters obtain from 4.2. With stacking generalization, we stack Random Forest, Ridge Kernel, Gaussian process together and use Random Forest as our final estimator. The results are in the below table:

	Random Forest		Ridge Kernel		Gaussian process		Ensemble NN	
	Train	**Test**	Train	Test	**Train**	Test	Train	Test
MAPE	0.196	**0.225**	0.216	0.233	**0.139**	0.383	0.227	0.227
RMSE	18.37	**24.32**	20.46	25.64	**13.25**	33.77	22.27	25.23
MAE	13.58	**17.67**	17.16	18.77	**9.81**	27.37	16.43	18.44

Table 7: Different model performance on train – test set

Random Forest, Ridge Kernel, Ensemble neural networks perform quite well on both the train and test set, while Gaussian process has the sign of overfitting. Gaussian process performs the best on the training set with very low MAPE = 0.139 but has very bad test results (MAPE = 0.338). In other three algorithms, Random Forest performs best on both the training set and the test set. Therefore, we can conclude that Random Forest is the best algorithm for the task of house price prediction in Hanoi.

5. Conclusions:

In this project, we have applied some Machine Learning methods to solve an intriguing problem of predicting real estate price in Hanoi. We believe that the results have showed the power of Machine Learning in applying to solve the real- world problem. We also acknowledge that these methods used in this project are just a small attempt to tackle this dilemma. There are many more waited to be discovered.

References

[1] MIT 6.036 Machine Learning 2019 Course notes.

[2] Diederik P. Kingma, Jimmy Ba. Adam: A Method for Stochastic Optimization. ICLR (2015).

[3] CE Rasmussen, CKI Williams. Gaussian processes in machine learning (MIT Press) (2014)

[4] Cornell CS4780 Introduction to Machine Learning for Intelligent Systems 2018 Course notes.

[5] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra & Kilian Q. Weinberger Deep Networks with Stochastic Depth (2016) arXiv 1603.09382.

[6] Kenvin P. Murphy. (2022). Probabilistic Machine Learning: An introduction. MIT press

[7] Khoat. Than. (2022). Decision Tree and Random Forest. Hanoi University of Science and Technology.https://users.soict.hust.edu.vn/khoattq/lectures/IT3190E-131679-ML/L4-Random-forests.pdf.