Building a Real Estate Price Prediction Model with Data Science

Introduction

In this tutorial series, we will delve into the steps and challenges that data scientists encounter in their day-to-day work, focusing on a real-life data science project. Imagine yourself as a data scientist working for a prominent real estate company like Zillow in the US or Magicbricks in India. Your business manager has approached you with a task to develop a model that can predict property prices based on various features such as square footage, number of bedrooms, bathrooms, and location. To make the project more engaging, we will also create a website using HTML, CSS, and JavaScript that enables users to perform home price predictions.

Project Architecture

Let's begin by outlining the architecture of our project. We will start by obtaining a home price dataset from Kaggle, specifically for the city of Bangalore in India. Using this dataset, we will build a machine learning model that incorporates several data science concepts, including data cleaning, feature engineering, dimensionality reduction, and outlier removal. Once the model is developed, we will export it to a pickle file. Additionally, we will create a Python Flask server that can utilize this pickle file to make price predictions. The Flask server will expose various HTTP

endpoints to handle different requests. To complement the server, we will create a user interface using HTML, CSS, and JavaScript, enabling users to interact with the model.

Tools and Technology

To accomplish our project objectives, we will utilize the following tools and technologies:

- Python: We will use Python as our programming language, known for its extensive libraries and ease of use in data science.
- Pandas: This library will assist us in data cleaning and manipulation tasks.
- Matplotlib and Seaborn: These data visualization libraries will enable us to gain insights from the dataset.
- Scikit-learn (SKlearn): SKlearn will be instrumental in building our machine learning model.
- Python Flask: We will employ Flask, a micro web framework, to develop the back-end server that interacts with the machine learning model.
- HTML, CSS, and JavaScript: These front-end technologies will help us create a user-friendly website interface.

Project Execution

Now that we have a clear understanding of the project's architecture and the tools we will employ, let's delve into the execution of the project. The following steps outline the process:

1. Acquire the home price dataset from Kaggle for the city of Bangalore.
2. Perform data cleaning to handle missing values, outliers, and

other inconsistencies in the dataset.
3. Employ feature engineering techniques to extract meaningful information from the available features.
4. Utilize dimensionality reduction methods, such as principal component analysis (PCA), to reduce the dataset's complexity.
5. Build a machine learning model using SKlearn, training it on the preprocessed dataset.
6. Evaluate the model's performance using appropriate metrics and fine-tune it if necessary.
7. Export the trained model to a pickle file for later use.
8. Develop a Python Flask server that exposes HTTP endpoints to handle price prediction requests.
9. Create a user interface using HTML, CSS, and JavaScript to interact with the Flask server.
10. Implement functionality on the website to make HTTP GET and POST calls for price predictions.
11. Deploy the website and Flask server on a suitable hosting platform to make it accessible to users.

Conclusion

Embarking on this real estate price prediction project will provide you with valuable insights into the life of a data scientist working for a major company. Throughout the project, you will encounter various challenges and learn essential data science concepts and techniques. By utilizing Python, Pandas, SKlearn, Flask, and front-end technologies, you will gain hands-on experience in building a machine learning model and developing a functional website. This project promises to be both educational and exciting, so let's dive right in and start building our real estate price prediction model.

00:00-15:16

Data Cleaning Techniques in Pandas: A Comprehensive Guide

Introduction

Data cleaning is a crucial step in the data analysis process. In this article, we will explore various techniques for data cleaning using the Pandas library in Python. We will walk through the process of downloading a dataset, loading it into a Pandas DataFrame, and applying different data cleaning techniques to ensure the dataset is ready for further analysis.

Downloading and Loading the Dataset

To follow along with this tutorial, you can download the dataset from the provided link in the video description. Once downloaded, we can load the dataset into a Pandas DataFrame using the `read_csv()` function. This function reads the CSV file and stores the data in a tabular format.

```
import pandas as pd

# Load the dataset into a Pandas DataFrame
df = pd.read_csv('dataset.csv')
```

Examining the Dataset

Before diving into data cleaning, let's first examine the dataset to get a better understanding of its structure. The dataset contains various independent variables such as area type, location, size, and total square foot area. The dependent variable we are trying to predict is the price of the properties listed in the dataset.

```
# Display the shape of the dataset (number of rows and columns)
print(df.shape)

# Display a sample of the dataset
print(df.head())
```

Handling Missing Values

Dealing with missing values is an important part of data cleaning. To identify missing values in the dataset, we can use the `isnull()` function, which returns a boolean DataFrame indicating the presence of missing values. We can then use the `dropna()` function to remove the rows containing missing values.

```
# Check for missing values
print(df.isnull().sum())

# Drop rows with missing values
df.dropna(inplace=True)
```

Data Transformation and Cleaning

In this section, we will perform further data cleaning and transformation operations on the dataset.

Dropping Unnecessary Columns

Some columns may not contribute significantly to the analysis or modeling process. In such cases, we can drop those columns from the DataFrame using the `drop()` function.

```
# Drop unnecessary columns
df.drop('availability', 'society', 'area type',
axis=1, inplace=True)
```

Handling Categorical Variables

In our dataset, the "size" column contains values such as "2 BHK" or "4 BHK." To handle these values, we can create a new column called "BHK" and extract the numeric part from the "size" column using string manipulation techniques.

```
# Create a new column for the number of bedrooms
(BHK)
df'BHK' = df'size'.apply(lambda x:
int(x.split()0))
```

Handling Irregularities in the Dataset

Sometimes, the dataset may contain irregularities or errors that need to be addressed. For example, the "total square foot" column may include ranges or units other than square feet. We can apply specific data cleaning functions to handle such cases.

```
# Function to convert range values into average
numbers
def convert_range_to_avg(value):
    if '-' in value:
        tokens = value.split('-')
        return (float(tokens0) + float(tokens1))
/ 2
    elif 'sqft' in value:
        return float(value.replace('sqft', ''))
    else:
        return None

# Apply the function to the \"total square foot\"
column
df'total_sqft' = df'total
sqft'.apply(convert_range_to_avg)
```

Conclusion

Data cleaning is a crucial step in preparing a dataset for analysis. In this article, we covered various data cleaning techniques using the Pandas library. We explored methods to handle missing values, transform and clean data, and remove unnecessary columns. By applying these techniques, we can ensure that our dataset is ready for further analysis and modeling.

In the next article, we will dive into feature engineering and dimensionality reduction techniques, which will further enhance our dataset for advanced analysis. Stay tuned!

00:00-08:22

Feature Engineering and Dimensionality Reduction Techniques

In this article, we will explore various feature engineering and dimensionality reduction techniques to enhance our data analysis process. We will begin by examining the steps involved in creating new features and then delve into reducing the dimensions of our dataset. These techniques will aid us in outlier detection and removal, ensuring the reliability of our analysis.

Creating the Price per Square Feet Feature

To start, let's consider a real estate dataset where we have a data frame with various columns. One important feature in the real estate market is the price per square feet, which can be useful for outlier detection. We will create a new column, "Price per Square Feet," by dividing the price column by the square foot area column. This will provide us with a valuable feature for later stages of analysis.

```
df5'Price per Square Feet' = df5'Price' /
df5'Square Foot Area'
```

With the addition of the new feature, our data frame now includes the "Price per Square Feet" column. This column will contribute to identifying and addressing outliers in subsequent steps.

Exploring the Location Feature

Next, let's focus on the location column in our dataset. It is a categorical feature that contains information about the various locations. We aim to determine the number of unique locations and the corresponding number of rows in our dataset.

```
unique_locations = df5'Location'.unique()
num_locations = len(unique_locations)
num_rows = df5.shape0
```

After executing the above code, we find that we have a large number of locations, with a total of 1,300 unique locations in our dataset. However, dealing with such a large number of locations can pose challenges, specifically regarding the dimensionality of the data.

Addressing the Dimensionality Curse

High dimensionality, also known as the "dimensionality curse," can negatively impact our data analysis. When we convert categorical features like location into dummy columns using one-hot encoding, we end up with an overwhelming number of columns. In this case, we would have 1,300 columns, which is impractical and can lead to computational inefficiencies.

To combat this issue, we can introduce the concept of an "other" category. This category encompasses locations that have a limited number of data points, such as only one or two instances. By identifying these locations, we can reduce the dimensions of our dataset effectively.

To begin, we need to determine the number of data points available for each location. Let's clean the location column by removing any extra spaces using a lambda function:

```
df5'Location' = df5'Location'.apply(lambda x:
x.strip())
```

Now, we can calculate the statistics on location by grouping the data frame based on the location column and aggregating the count for each location. This will provide us with the number of data points available for each location.

```
location_stats =
df5.groupby('Location')'Location'.count()
```

Sorting the resulting statistics in descending order will reveal the locations with the maximum number of data points. We can use this information to establish a threshold, such as considering locations with less than ten data points as "other" locations.

```
location_stats =
location_stats.sort_values(ascending=False)
threshold = 10
other_locations = location_statslocation_stats <
threshold
```

By applying the condition mentioned above, we find that there are 1,052 locations out of the original 1,293 that have less than ten data points. These locations will be consolidated into a general category called "other."

```
df5'Location' = df5'Location'.apply(lambda x:
'other' if x in other_locations else x)
```

After transforming the data frame, we observe that the number of unique locations reduces to 242. This reduction in dimensions will significantly aid us in subsequent data analysis steps.

Conclusion

In this article, we covered essential techniques for feature engineering and dimensionality reduction. We explored the creation of the "Price per Square Feet" feature, which facilitates outlier detection. Additionally, we addressed the issue of high dimensionality by introducing an "other" category for locations with a limited number of data points. By implementing these techniques, we successfully reduced the dimensionality of our dataset, enhancing the efficiency and accuracy of our analysis. In the next article, we will dive into outlier detection and removal, further refining our data analysis process.

00:00-19:26

Outlier Detection and Removal Techniques in Data Analysis

Introduction

Outliers are data points that deviate significantly from the majority of the data in a dataset. They can be caused by data errors or represent extreme variations in the data. In this article, we will explore different techniques for detecting and removing outliers to ensure the integrity and accuracy of our dataset.

Standard Deviation Method

One commonly used technique for outlier detection is the standard deviation method. By calculating the standard deviation of a feature, we can identify data points that lie beyond a certain threshold from the mean. These points can be considered outliers and removed from the dataset.

Domain Knowledge Approach

Another approach for outlier detection is using domain knowledge. In certain domains, there are specific rules or expectations that can help identify outliers. For example, in the real estate domain, a two-bedroom apartment would typically have a minimum square footage threshold. If any data points fall below this threshold, they can be considered outliers and removed from the dataset.

Data Cleanup Process

To illustrate the outlier removal process, let's consider a scenario where we have a dataset of real estate properties. We will go through the steps of outlier detection and removal using various techniques.

Step 1: Detecting Square Footage Outliers

We consult with a real estate expert to determine the typical square footage per bedroom ratio. Based on their input, we identify data points where the square footage per bedroom falls below the expected threshold. These data points are considered outliers and need to be removed from the dataset.

Step 2: Removing Square Footage Outliers

We create a new dataframe and filter out the identified outliers based on the square footage per bedroom criteria. After removing these outliers, we verify the number of remaining data points.

Step 3: Detecting Price per Square Foot Outliers

Next, we examine the price per square foot feature. We look for properties with abnormally high or low price per square foot values, which can be considered outliers. By using statistical measures such as mean and standard deviation, we can filter out the extreme cases.

Step 4: Removing Price per Square Foot Outliers

We develop a function that removes outliers based on the standard deviation approach. This function groups the data by location, calculates mean and standard deviation values, and filters out data points beyond one standard deviation from the mean. By applying this function to the dataframe, we remove the price per square foot outliers.

Step 5: Analyzing Bedroom vs. Price

We compare the property prices for two-bedroom and three-bedroom apartments with the same square footage. If we observe that the two-bedroom apartments have higher prices, it could indicate outliers or anomalies. We create scatter plots to visualize these cases and identify any outliers that need to be removed.

Step 6: Removing Bedroom vs. Price Outliers

We develop a function that removes outliers based on the comparison of bedroom and price values. By calculating the mean of one-bedroom apartments and filtering out two-bedroom apartments with prices below the mean, we can remove these outliers.

Step 7: Analyzing Bathroom Feature

We examine the bathroom feature and notice that some properties have an unusually high number of bathrooms. We consult with our business manager and establish a criterion for identifying bathroom outliers. In this case, we remove properties with a number of bathrooms greater than the number of bedrooms plus two.

Step 8: Data Cleanup and Preparation

After removing all the identified outliers, we drop unnecessary features such as price per square foot and size from the dataset. This step helps prepare the dataset for machine learning training.

Conclusion

Outlier detection and removal are crucial steps in data analysis to ensure the quality and reliability of the dataset. By applying techniques such as standard deviation, domain knowledge, and statistical analysis, we can identify and eliminate outliers that could potentially affect our analysis or machine learning models. Through careful data cleanup and preparation, we can create a clean and accurate dataset for further analysis or model training.

00:00-19:08

Building a Machine Learning Model with K-Fold Cross-Validation and Grid CV

Introduction

In the previous tutorial, we focused on cleaning up our data frame to prepare it for model building. In this tutorial, we will dive into the process of building a machine learning model. Additionally, we will utilize K-Fold cross-validation and Grid CV to identify the best algorithm and parameters. Our dataset contains a location column, and since machine learning models cannot interpret text data, we need to convert this categorical information into numerical data. One common method for achieving this is one-hot encoding, also known as dummies. We will leverage the pandas `get_dummies` function to perform one-hot encoding.

One-Hot Encoding

To convert the location column into numerical data, we will use the `get_dummies` function from the pandas library. The process involves creating new columns for each unique location value and setting the corresponding values to 1 or 0. This straightforward encoding method allows us to represent categorical information

numerically. Once the dummy columns are created, we will store them in a separate data frame and append them to our main data frame.

Model Building

After performing the necessary data processing steps, we can begin building our machine learning model. First, let's examine the shape of the data frame, which consists of 7,000 rows and 245 columns. To train our model, we need to separate the independent variables (X) from the dependent variable (price, Y). We will drop the price column from the data frame to obtain the X variable.

Next, we will split our dataset into training and test sets using the `train_test_split` method from the `sklearn.model_selection` module. We will allocate 20% of the samples for testing and use the remaining 80% for model training.

We will create a linear regression model and fit it using the training data. Once the model is trained, we can evaluate its performance by calculating the score. In this case, our linear regression model achieves a score of 84%, which is quite decent. However, to ensure we have the best possible model, we will explore other regression techniques.

K-Fold Cross-Validation

To assess the performance of different regression techniques, we will employ K-Fold cross-validation. This technique involves dividing the dataset into K subsets or folds, training the model on K-1 folds, and evaluating it on the remaining fold. We import the necessary

modules and create a ShuffleSplit object to randomize the samples within each fold. By using cross-validation, we obtain scores consistently above 80%.

Grid Search CV

Although linear regression performs well, we want to try other regression algorithms to determine the best score. To accomplish this, we use Grid Search CV, which performs an exhaustive search over specified parameter values for each algorithm. By specifying the algorithms and their respective parameter grids, we can identify the best algorithm and its corresponding parameters. In this case, linear regression emerges as the winner with the highest score. The best parameter for linear regression is `normalized=False`.

Model Prediction

With our linear regression model selected, we can proceed to make property price predictions. We create a function, `predict_price`, that takes inputs such as location, square foot, bath, and BHK (bedrooms, hall, kitchen). By identifying the appropriate column index for the given location, we can set the value to 1 and predict the price based on the provided inputs.

Exporting the Model

Now that our model building procedure is complete, we need to export the artifacts required for deployment on our Python Flask server. We export the trained model using the pickle module, which allows us to serialize Python objects. Additionally, we export the column information as a JSON file. The model file size is small because it only stores coefficients, intercepts, and other

parameters, without including the actual data. The JSON file contains the lowercase column names, ensuring consistency for making predictions in the future.

Conclusion

As data scientists, we go through various iterations, trying different models, performing grid search CV, and cleaning the data to arrive at an optimal model. We have successfully built a linear regression model with a score of 84%, demonstrating its suitability for predicting property prices. In the next tutorial, we will focus on developing a Python Flask server that utilizes the exported model and artifacts.

00:00-21:10

Building a Python Flask Server for Home Price Prediction

Introduction

In this article, we will discuss how to build a Python Flask server that can handle HTTP requests from a user interface (UI) and predict home prices. The Flask server will serve as the backend for our UI application. We will guide you through the process of setting up the server, creating the necessary routines, and testing the endpoints.

Setting up the Project

To begin, make sure you have PyCharm Community Edition installed, which can be downloaded for free from the JetBrains website. Once installed, open the project in PyCharm. The project directory should have three subfolders: client, server, and model. The client folder will house the UI application, the server folder will contain the Flask server code, and the model folder will store the notebook and exported artifacts (saved model and columns.json) from previous tutorials.

Creating the Flask Server

1. In the server folder, create a new file called server.py.
2. Import the Flask module in the server.py file. Flask allows us

to write Python services that can handle HTTP requests.
3. Configure the Python interpreter to use Anaconda (or any other preferred interpreter) as it comes with Flask pre-installed.
4. Create a Flask app using the `app = Flask(__name__)` line.
5. In the main function, run the application using `app.run()`.
6. Write a simple "hello" route by adding the following code:

```
@app.route('/hello')
def hello():
    return 'Hi'
```

This route will return the response 'Hi' when accessed.

Testing the Flask Server

1. Run the server by executing `python server.py`.
2. Copy the URL displayed in the terminal and paste it into your browser.
3. Append `/hello` to the URL and press Enter.
4. You should see the response 'Hi' displayed in your browser, indicating that the Flask server is up and running.

Retrieving Location Names

Next, we will create a routine to retrieve location names from a JSON file. This routine will be used to populate a dropdown menu in the UI application.

1. Create a subdirectory called "artifacts" within the server directory.
2. Copy the exported artifacts (model and columns.json) into the

artifacts directory.
3. Read the columns.json file to extract the location names.
4. Create a new file called util.py in the server directory to store utility functions.
5. In util.py, define a function called `get_location_names()` to retrieve the location names.
6. Return the location names using the `jsonify()` method.
7. Test the `get_location_names()` function to ensure it returns the expected results.

Estimating Home Prices

We will now create a function to estimate home prices based on location, square footage, number of bedrooms, and number of bathrooms.

1. Define a function called `get_estimated_price(location, sq_ft, bhk, bathroom)` that takes these parameters.
2. Load the saved artifacts (model and columns.json) into memory.
3. Create a numpy array with zeros to hold the input values.
4. Find the index corresponding to the provided location in the columns list.
5. Set the corresponding element in the numpy array to 1 and the remaining elements to 0 using one-hot encoding.
6. Use the loaded model to predict the home price by calling the `predict()` method.
7. Round the predicted price to two decimal places.
8. Return the estimated price.

Implementing the Predict Home Price Endpoint

In the server.py file, create a new endpoint called `/predict_home_price` using the HTTP POST method.

1. Import the `request` class to retrieve the input parameters from the request form.
2. Get the input values (total square footage, location, BHK, and bathroom) from the request form.
3. Call the `get_estimated_price()` function with the input parameters to get the estimated price.
4. Return the estimated price as the response.

Testing the Endpoints

To test the endpoints, we can use an application like Postman.

1. Open Postman and select the GET method.
2. Enter the URL with the endpoint `/get_location_names` and click Send.
3. The response will contain a list of all the location names.
4. Switch to the POST method and set the URL to `/predict_home_price`.
5. Specify the input parameters (total square footage, location, BHK, and bathroom) in the form data.
6. Click Send, and the response will contain the estimated home price.

By following these steps, you can successfully build a Python Flask server that can handle HTTP requests and predict home prices based on user input. With the Flask server in place, you can now

proceed to develop the UI for a complete application.

In the next article, we will guide you through building the UI, which will integrate with the Flask server to provide a seamless user experience.

00:00-13:18

Building a Website with HTML, CSS, and JavaScript

Introduction

In the previous video, we developed a Python Flask server that serves as the backend for our website. Now, in this tutorial, we will build the frontend of our website using HTML, CSS, and JavaScript. We will create three files: `app.html`, `app.css`, and `app.js`. These files will define the structure, style, and dynamic behavior of our web application.

HTML Structure

In the `app.html` file, we will define the structure of our user interface (UI). The HTML code will consist of two main sections: the head and the body. The head section includes the title of the application and the import of necessary files such as jQuery, `f.js`, and `f.css`. The body section contains the main elements of our application, including an input for area, buttons for BHK (bedroom, hall, kitchen) selection, a button bar for bathroom options, and a dropdown for location selection.

Styling with CSS

To style our website, we will use CSS. We have defined several CSS classes in the `app.css` file that correspond to the UI elements mentioned earlier. These classes define properties like background color, font style, and other visual aspects of the application.

Running the Application

To see our UI in action, we can open the `app.html` file in a web browser. At this point, the locations in the dropdown are still hard-coded. We will dynamically populate the dropdown using JavaScript in the next step.

Communicating with the Backend using JavaScript

In the `app.js` file, we will write JavaScript code to communicate with the backend server and retrieve data. First, we will implement a function to load the locations. Using JavaScript's `window.onload` event, we can make an HTTP GET request to the server to fetch the list of locations. We will then iterate through the locations and dynamically add them to the dropdown.

Next, we will implement the `onEstimatePriceClick` function, which will be triggered when the user clicks the "Estimate Price" button. This function will retrieve the values of the UI elements (area, BHK, bathroom, and location) using helper functions `getBHKValue` and `getBathValue`. With these values, we will make an HTTP POST request to the server's "predict home price" endpoint. The response will contain the estimated price, which we will display on the UI.

Testing the Application

After refreshing the page, we can test the application by selecting a location and clicking the "Estimate Price" button. The estimated price will be displayed based on the selected features. Different locations and property configurations will yield different prices.

Deployment and Conclusion

At this point, our application is working fine locally. However, if we want to deploy it in a production environment, additional steps and procedures need to be followed. In a typical workflow, a data scientist builds models, a Python Flask server is developed, and a separate software engineering team handles the UI and backend application. However, it's beneficial for data scientists to have some knowledge of web development to test their models effectively.

In summary, this tutorial series has provided insights into how a data scientist working for a large organization builds an application from end to end. We covered data cleaning, dimensionality reduction, feature engineering, model building, creating a Python Flask server, and developing the UI using HTML, CSS, and JavaScript. We hope you found this series informative and useful for your journey as a data science enthusiast.

Remember to give a thumbs up and share this series with others who may find it helpful. Thank you for watching!

Bye!

00:00-28:35

Deploying a Machine Learning Model to Production: A Comprehensive Guide

Introduction

In this article, we will walk through the process of deploying a machine learning model to production. Specifically, we will focus on deploying a Bangalore home price prediction website to Amazon EC2, a cloud computing service provided by Amazon Web Services (AWS). By the end of this tutorial, you will have a fully functional website accessible through a URL, which can be mapped to your own domain. Let's dive into the details and explore the step-by-step process of deploying your code end-to-end.

Architecture Overview

Before we proceed with the deployment, let's take a brief look at the architecture of our application in the Amazon EC2 instance. The key components involved in the deployment are as follows:

1. Nginx: We will use Nginx as our lightweight web server. It will handle the incoming HTTP requests and serve the static files of our website.

2. Python Flask Server: Our Python Flask server will run on the same EC2 instance. It will handle the API requests and utilize the saved machine learning model to make predictions.

3. Reverse Proxy Setup: To route the API requests from Nginx to our Flask server, we will configure a reverse proxy setup on Nginx.

With this architecture in mind, we can now proceed to set up Nginx in the development environment and then replicate the setup on an Ubuntu server in our EC2 instance.

Setting Up Nginx on Windows

1. Download Nginx by visiting the official website or searching for it on Google.

2. Unzip the downloaded file and copy the extracted folder to a convenient location on your system (e.g., C:\Program Files).

3. Locate the Nginx configuration file (nginx.conf) within the Nginx folder and open it.

4. Verify that the Nginx server is running correctly by double-clicking the executable file. This will start the web server, and you should be able to access the default Nginx welcome page by navigating to `localhost` in your web browser.

5. Edit the configuration file (nginx.conf) and update the root directory to point to your Bangalore home prices page. Replace the existing root directory path with the path to your website's HTML file.

6. Save the configuration file and restart the Nginx server.

7. Verify the changes by refreshing the web page. Now, when you access `localhost`, your Bangalore home prices page should be loaded instead of the default Nginx welcome page.

8. Test the functionality of the website, including the prediction feature, by interacting with it in the browser. You can use the browser's developer tools (press F12) to monitor incoming and outgoing network requests for debugging purposes.

Configuring the Reverse Proxy for Flask Server

To enable the communication between Nginx and the Flask server running on port 5000, we need to set up a reverse proxy on Nginx. Follow these steps to configure the reverse proxy:

1. Open the Nginx configuration file (nginx.conf) mentioned earlier.

2. Add the following configuration section to the file:

```
location /api/ {
    proxy_pass http://localhost:5000/;
}
```

This configuration ensures that any HTTP request with a path starting with `/api/` is routed to the Flask server running on port 5000.

3. Save the configuration file and restart the Nginx server.

Running the Flask Server

Before we proceed to deploy our application on the cloud, we need to run the Flask server locally to ensure everything is working as expected. Follow these steps to run the server:

1. Open a command prompt or terminal and navigate to the server directory of your project.

2. Run the command `python server.py` to start the Flask server on port 5000.

3. Verify that the server is running correctly by refreshing the website. The website should now be fully functional, and the API requests should be successfully routed to the Flask server.

Deploying to Amazon EC2

To deploy our application on Amazon EC2, follow these steps:

1. Sign in to the AWS Management Console using your Amazon account.

2. Search for and select the Amazon EC2 service.

3. Click on "Launch Instance" to create a new virtual machine (EC2 instance).

4. Select the desired machine configuration, such as the operating system (e.g., Ubuntu) and the instance type. Choose the free tier if applicable.

5. Follow the default settings provided by Amazon, including storage configuration.

6. Configure the security group to allow incoming HTTP requests on port 80. This will ensure that your web server is accessible from the outside world.

7. Review the instance details and launch the instance.

8. Create a new key pair or select an existing one for secure SSH access to the EC2 instance.

9. Download the key pair file (.pem) and store it in a secure location on your local machine.

10. Launch the instance and wait for it to be initialized.

11. Connect to the EC2 instance using SSH. You can use a tool like Git Bash to establish an SSH connection by specifying the key pair file and the public DNS of the instance.

12. Once connected to the EC2 instance, use a tool like WinSCP to copy your codebase from your local machine to the EC2 instance.

13. Install Nginx on the EC2 instance by running the necessary commands to update the packages and install Nginx.

14. Update the Nginx configuration files (nginx.conf and bhp.conf) on the EC2 instance to reflect the correct paths and settings.

15. Restart the Nginx server to apply the changes.

16. Start the Flask server on the EC2 instance using the appropriate command.

17. Verify that the website is accessible by visiting the public DNS of your EC2 instance in a web browser.

18. Test the website's functionality, including the prediction feature, to ensure everything is working as expected.

Conclusion

Congratulations! You have successfully deployed your machine learning model to production by hosting a Bangalore home price prediction website on Amazon EC2. Throughout this tutorial, we covered the step-by-step process of setting up Nginx, configuring the reverse proxy for the Flask server, running the server locally, and deploying the application to the cloud. Remember to practice and experiment on your own to gain a deeper understanding of the deployment process. Feel free to refer to the provided GitHub repository for the code and additional instructions. If you encounter any issues or have any questions, don't hesitate to ask in the comments. Happy deploying!