
Universidade Federal da Paraíba
Centro de Ciências Exatas e da Natureza
Departamento de Estatística

(Escolher no final)

Gabriel de Jesus Pereira

janeiro, 2025

Gabriel de Jesus Pereira

(Escolher no final)

Trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Estatística, do Departamento de Estatística, do Centro de Ciências Exatas e da Natureza, da Universidade Federal da Paraíba, como requisito para obtenção do título de Bacharel em Estatística.

Orientador: Prof. Dr. Pedro Rafael Diniz Marinho

**João Pessoa
janeiro, 2025**

Sumário

1	Introdução	7
1.1	Objetivos	10
1.1.1	Objetivo Geral	10
1.1.2	Objetivos Específicos	10
1.2	Organização do Trabalho	10
2	Recursos Computacionais	12
2.1	Recursos utilizados para a coleta de dados	12
2.2	Recursos utilizados para modelagem e visualização dos resultados	15
2.3	Recursos utilizados para a criação da aplicação final	17
2.4	Recursos utilizados para escrita de código e de documento	18
3	Algoritmos de Aprendizado de Máquina	20
3.1	Árvore de decisão	20
3.2	Métodos Ensemble	23
3.2.1	Bagging	24
3.2.2	Random Forest	25
3.2.3	Boosting Trees	26
3.2.4	Stacked generalization	28
3.2.5	Gradient Boosting	28
4	Metodologia	31
4.1	Obtenção dos dados	31
4.2	Análise exploratória de dados	32
4.3	Construção do modelo	32
4.3.1	Etapas de pré-processamento	33
4.3.2	Validação cruzada	34
4.4	Otimização de hiperparâmetros	36
4.4.1	Tree-Structured Parzen Estimator	37
4.4.2	Otimização de hiperparâmetros com optuna	38
4.5	Interpretação dos algoritmos de aprendizagem de máquina	40
4.5.1	Individual Conditional Expectation (ICE)	40
4.5.2	Local interpretable model-agnostic explanations (LIME)	42
4.5.3	Shapley Additive Explanations (SHAP)	42
5	Resultados	46
5.1	Análise exploratória de dados	46
5.2	Tunagem dos modelos	50
5.2.1	Tunagem da Random Forest	50
5.2.2	Tunagem do Gradient Boosting	52
5.2.3	Tunagem do LGBM	52
5.2.4	Tunagem do XGBoost	55

5.3	Resultados dos modelos	57
5.4	Impacto e importância das variáveis na predição	59
6	Conclusão	62

Lista de Figuras

3.1	Exemplo de estrutura de árvore de regressão. A árvore tem cinco folhas e quatro nós internos.	21
4.1	Visualização de K-Fold com 20 folds.	35
5.1	Proporção de valores ausentes por variáveis	46
5.2	Variação da média do valor do m^2 dos imóveis de João Pessoa, área de estudo.	47
5.3	Distribuição das variáveis numéricas.	48
5.4	Comparação entre distribuição dos valores dos imóveis antes e depois da transformação logarítmica.	49
5.5	Gráfico de correlação de Spearman das variáveis independentes.	49
5.6	Resultados da tunagem da Random Forest.	51
5.7	Resultados da tunagem do Gradient Boosting	53
5.8	Resultados da tunagem do LGBM	54
5.9	Resultados da tunagem do XGBoost	56
5.10	Valores previstos em função dos observados do algoritmo Random Forest e Gradient Boosting, respectivamente.	57
5.11	Valores previstos em função dos observados do algoritmo Light Gradient Boosting e Extreme Gradient Boosting, respectivamente.	58
5.12	Valores previstos em função dos observados do algoritmo Stacking	58
5.13	Gráfico de ICE e PDP.	60
5.14	Impacto e importância das variáveis na predição a partir do método SHAP.	60
5.15	Gráfico de dependência de valores SHAP para variáveis binárias.	61

Lista de Tabelas

5.1	Métricas obtidas de cada algoritmo	50
5.2	Melhores hiperparâmetros para Random Forest	52
5.3	Melhores hiperparâmetros para Gradient Boosting	52
5.4	Melhores hiperparâmetros para Light Gradient Boosting	55
5.5	Melhores hiperparâmetros para Extreme Gradient Boosting	55

Lista de Algoritmos

3.1	Algoritmo para crescer uma árvore de regressão	23
3.2	Algoritmo de uma Random Forest para regressão ou classificação	26
3.3	Método Boosting aplicado a árvores de regressão	27
3.4	Gradient Tree Boosting	29

Agradecimentos

Resumo

A história do Brasil, assim como a de muitos outros países, foi marcada por uma predominância rural durante grande parte de sua trajetória, com apenas 31% da população vivendo em áreas urbanas em 1940. Contudo, esse cenário mudou rapidamente e, em 1970, mais da metade da população já residia em cidades (WAGNER; WARD, 1980). O processo de urbanização acelerado e desordenado resultou no crescimento desorganizado das grandes metrópoles, agravando problemas urbanos como o déficit habitacional. Com o aumento das áreas urbanizadas, a demanda por habitação tornou-se cada vez maior, levando à criação de medidas para mitigar esse déficit. Nesse contexto, a Lei nº 4.380, de 21 de agosto de 1964, instituiu pela primeira vez no país um mecanismo de crédito habitacional, sendo fundamental para articular a oferta e a demanda de recursos destinados ao setor habitacional. Essa legislação resultou na criação do Sistema Financeiro de Habitação (SFH), que trouxe inovações como a correção monetária, o Banco Nacional da Habitação (BNH) e as Sociedades de Crédito Imobiliário (SCI). Apesar dos avanços do SFH, o aumento da inflação na década de 1980 impactou seu funcionamento, agravado por medidas governamentais para conter a inflação. Como evolução desse sistema, foi criado o Sistema de Financiamento Imobiliário (SFI), que não substituiu o SFH, mas introduziu um modelo moderno de financiamento habitacional, fundamentado na securitização de créditos imobiliários, maior segurança jurídica dos contratos e captação de recursos diretamente no mercado por meio de operações realizadas por entidades autorizadas. A dinâmica do tempo e a modernização do mercado imobiliário trouxe impacto significativo para diversas cidades brasileiras, incluindo João Pessoa, capital da Paraíba, que se tornou um dos destinos mais procurados para viver e visitar. A cidade destacou-se como uma das capitais com maior valorização acumulada de imóveis residenciais nos últimos 12 meses, até novembro de 2024, apresentando uma variação de 16,13%. O crescimento do mercado imobiliário em João Pessoa tem gerado novas demandas, incluindo a necessidade de métodos mais precisos para precificação de imóveis. Dessa forma, este trabalho teve como objetivo desenvolver uma modelagem estatística para precificação de valores imobiliários na cidade de João Pessoa, utilizando dados coletados por meio de raspagem de sites especializados. Além da modelagem, buscou-se também analisar o impacto das variáveis utilizadas na estimativa dos preços dos imóveis, empregando técnicas de aprendizagem de máquina como base metodológica. O modelo final foi construído utilizando o algoritmo Stacking e apresentou resultados satisfatórios, com um R^2 de 87,19%, erro percentual absoluto médio (MAPE) de 1,357% e raiz do erro quadrático médio (RMSE) de 0,28473. A análise das variáveis revelou que a área do imóvel foi o fator mais determinante na precificação, seguida pela quantidade de vagas de garagem, valor médio de aluguel do bairro, coordenadas geográficas, número de quartos e banheiros, e área média de aluguel do bairro. Por fim, foi desenvolvida uma aplicação prática para facilitar a precificação e a análise de imóveis em João Pessoa, contribuindo para maior precisão no processo de avaliação imobiliária.

Palavras-chave: Aprendizagem de máquina, mercado imobiliário, modelagem estatística, João Pessoa.

1 Introdução

Assim como em todos os países, a maior parte da história do Brasil foi marcada por uma predominância rural. Em 1940, apenas 31% da população vivia em áreas urbanas. No entanto, esse número cresceu rapidamente, ultrapassando 50% em 1970 (WAGNER; WARD, 1980). Essa tendência se manteve nas décadas seguintes, até que, em 2014, 85,1% da população brasileira já residia em regiões urbanas.

No Brasil, o processo de urbanização ocorreu de forma tardia e desordenada, quando comparado com os países pioneiros da Revolução Industrial, que tiveram todo um planejamento para suportar a grande transição demográfica que a população estava passando naquele momento. Um dos principais fatores que impulsionaram essa rápida urbanização, no Brasil, foram os chamados fatores de repulsão, característicos de nações em desenvolvimento. Esses fatores estão ligados às precárias condições de vida no meio rural, decorrentes da estrutura fundiária concentrada, dos baixos salários, desemprego e mecanização do campo. Como consequência, ocorreu o êxodo rural¹, que resultou no crescimento desordenado das grandes metrópoles e no agravamento de problemas urbanos, como déficit habitacional, precariedade nos serviços públicos e aumento das desigualdades sociais.

O êxodo rural no Brasil foi um processo gradual, mas ocorreu de forma acelerada quando comparado a outros países, atingindo seu auge entre 1960 e 1980. No entanto, foi durante o governo de Getúlio Vargas, na década de 1930, com o início do processo de industrialização, que surgiram as condições específicas para o aumento do êxodo rural, mas ainda era bastante limitado. Na década de 1950, o processo de urbanização se intensificou com a industrialização impulsionada pelo governo de Juscelino Kubitschek. Kubitschek implementou o Plano de Metas, que tinha como principal objetivo modernizar e desenvolver a infraestrutura do país. O Plano de Metas incluía investimentos estatais em setores estratégicos como agricultura, saúde, educação, energia, transporte, mineração e construção civil. Além dessas estratégias, o plano também previa a transferência da capital federal do Rio de Janeiro para Brasília, visando promover a ocupação e o desenvolvimento do interior do país.

Com os incentivos gerados pelo processo de industrialização e urbanização no Brasil, a demanda por habitação aumentou significativamente. Nesse cenário, surgiu a necessidade de se estabelecer, ainda que tardiamente, um sistema imobiliário no país. A criação desse sistema foi iniciada pela Lei nº 4.380, de 21 de agosto de 1964, que instituiu o Sistema Financeiro de Habitação (SFH). O sistema propiciou diversas inovações, como a correção monetária, a criação do Banco Nacional da Habitação (BNH) e as Sociedades de Crédito Imobiliário (SCI). Além disso, a legislação estabeleceu, pela primeira vez, um mecanismo de crédito habitacional capaz de articular a oferta e a demanda de recursos necessários para a realização de investimentos habitacionais.

¹Processo de transferência em larga escala da população do campo para as cidades.

O BNH tinha como objetivo oferecer incentivos ao mercado imobiliário, promovendo a poupança no país, além de atrair o mercado privado e regulamentar as condições de financiamento do Sistema Financeiro da Habitação (SFH), como garantias, prazos e taxas praticadas no sistema (ASSUMPÇÃO FILHO, 2011). As SCIs, subordinadas ao BNH, atuavam como agentes financeiros e eram restritas a operar exclusivamente no financiamento para construção, venda ou aquisição de bens destinados à habitação. Nesse período, foi criado o Fundo de Garantia do Tempo de Serviço (FGTS), uma forma de poupança compulsória que, junto à caderneta de poupança, tornou-se a principal fonte de financiamento habitacional no Brasil. As captações voluntárias realizadas por meio das cadernetas de poupança compõem o Sistema Brasileiro de Poupança e Empréstimo (SBPE), que reúne as instituições responsáveis pela captação de recursos livres, como as Sociedades de Crédito Imobiliário e as Associações de Poupança e Empréstimo.

A partir da década de 80 com o aumento da inflação no país, o SFH começa a ser afetado, principalmente com as ações tomadas pelos governos para combater a inflação. Em 1985, o reajuste das prestações foi de 112% e a inflação acumulada já alcançava os 246%, porcentual aplicado na correção dos saldos devedores (ABECIP, 2007a). Acuado pela inflação, os governos passados deram início a uma série de planos heterodoxos.

O Plano Cruzado, implementado em 1986, converteu o valor das prestações pela média dos 12 meses anteriores e, em seguida, congelou os reajustes pelos 12 meses seguintes. Essa medida atingiu a totalidade dos contratos e resultou em uma redução de cerca de 40% no valor das prestações. No entanto, a longo prazo, essa política afetou os contratos e os deixou mais caros. Nos anos seguintes, novas medidas foram implementadas. Em 1987 e 1989, as prestações foram congeladas temporariamente pelo Plano Bresser e pelo Plano Verão, respectivamente.

No Governo Collor, a medida mais prejudicial ao SFH foi o Plano Collor I, de 1990, que bloqueou todos os ativos financeiros e 60% do saldo das cadernetas de poupança, a principal fonte de arrecadação para o setor. O saldo da poupança na época correspondia a US\$ 30 bilhões. Dos 40% restantes, cerca de metade foi retirado pelos depositantes, uma vez que a população ficou sem dinheiro disponível para arcar com despesas correntes (ABECIP, 2015). Como consequência, o saldo das cadernetas de poupança foi drasticamente reduzido, atingindo aproximadamente US\$ 7 a US\$ 8 bilhões. Essa medida agravou intensamente a situação das instituições financeiras, que, de repente, ficaram sem passivo e ficaram com o ativo integral.

De acordo com a ABECIP (2007b), durante o período de auge do SFH, entre 1978 e 1982, o investimento habitacional por habitante manteve-se em torno de R\$ 500. Contudo, com as políticas econômicas adotadas pelos governos para combater a inflação, o investimento em habitação recuou e retornou ao patamar de R\$ 300. Com a criação do Plano Real, em 1994, observou-se uma pequena recuperação, embora ele permanecesse abaixo dos R\$ 500 no auge.

Como consequência da crise econômica que marcou o período entre 1980 e 1990, o arrocho salarial, a queda do poder aquisitivo, as altas taxas de juros e a inflação contribuíram significativamente para o aumento da inadimplência no SFH. Em 1994, a taxa de inadimplência estava próxima de 9%, enquanto em 2005 já havia se aproximado de 30% (COSTA FARIAS, 2010). Diante desse cenário, surgiram novos esforços e

iniciativas para contornar a crise e reformular o modelo de financiamento habitacional no Brasil.

Após a experiência acumulada com o SFH, a principal medida para reformular o modelo de financiamento habitacional no Brasil foi a criação do Sistema de Financiamento Imobiliário (SFI), que não significou o fim do SFH. O SFI foi instituído em 1997, pela Lei nº 9.514, como um complemento ao SFH.

Os principais fundamentos do SFI são a securitização dos créditos imobiliários e a maior segurança jurídica dos contratos. Diferentemente do SFH, o novo sistema capta recursos diretamente no mercado por meio de operações realizadas por entidades autorizadas, como caixas econômicas, bancos comerciais, bancos de investimento, bancos com carteira de crédito imobiliário, sociedades de crédito imobiliário, associações de poupança e empréstimo e companhias hipotecárias. Essas entidades podem aplicar os recursos utilizando instrumentos financeiros introduzidos com o SFI, tais como o Certificado de Recebíveis Imobiliários (CRI), a Letra de Crédito Imobiliário (LCI) e a Cédula de Crédito Imobiliário (CCI). A segurança jurídica dos contratos passou a ser garantida pela introdução da alienação fiduciária, um mecanismo que trouxe maior confiança e eficiência ao processo de financiamento imobiliário. Assim, o SFI representa a efetiva modernização do mercado imobiliário no País.

O mercado imobiliário tem demonstrado grande potencial em diversas capitais brasileiras, com destaque para João Pessoa, capital da Paraíba, que se tornou um dos destinos mais procurados para se viver devido à sua elevada qualidade de vida. João Pessoa foi fundada no atual bairro do Varadouro, às margens do Rio Sanhauá, por Martim Leitão e colonos vindos de Pernambuco, no dia 4 de novembro de 1585. Ao longo de sua história, a cidade recebeu três nomes antes de adotar sua nomenclatura atual, Nossa Senhora das Neves (1585), Frederickstadt (1634-1654) e Parahyba, nome que permaneceu até 1930. Em julho desse ano, a cidade foi renomeada como João Pessoa, em homenagem ao governador do estado da época. Durante o período colonial, a cidade era dividida em Cidade Alta e Cidade Baixa, interligadas por ladeiras (CAMPOS, 2014).

Atualmente, a cidade é detentora de um território de 211.475 km^2 e possui uma população de 833.932 habitantes, de acordo com o censo do IBGE 2022. Segundo o FIPE – FUNDAÇÃO INSTITUTO DE PESQUISAS ECONÔMICAS (2024), João Pessoa apresentou a maior valorização acumulada nos imóveis residenciais nos últimos 12 meses, até novembro de 2024, com uma variação de 16,13%. No mesmo período, considerando apenas o ano de 2024, a valorização acumulada foi de 15,15%, com o valor médio do metro quadrado residencial atingindo R\$ 6.867 no mês de novembro.

O crescimento do mercado imobiliário de João Pessoa trás consigo inúmeras necessidades de melhor especificação de imóveis. Avaliar corretamente o valor de um imóvel é crucial para uma série de finalidades, como a negociação justa entre compradores e vendedores e a determinação de valores tributários, como, por exemplo, o Imposto de Transmissão de Bens Imóveis (ITBI). Assim, a determinação do valor de um imóvel continua sendo um desafio, muitas vezes dependente de avaliações subjetivas ou métodos tradicionais que nem sempre refletem a complexidade dos fatores envolvidos.

Diante do cenário e dos desafios apresentados, este trabalho tem como objetivo analisar

e modelar os valores de diversos tipos de imóveis na cidade de João Pessoa. Além disso, por meio do estudo e dos dados utilizados em seu desenvolvimento, busca-se identificar os principais fatores que influenciam os preços dos imóveis e criar modelos preditivos capazes de auxiliar no processo de avaliação imobiliária, com potencial aplicação em atividades como o cálculo do ITBI. Além disso, o trabalho propõe analisar e descrever diferentes algoritmos de aprendizado de máquina, explorando suas características e desempenho no contexto proposto. Por fim, será desenvolvida uma aplicação computacional com múltiplas finalidades, visando servir como ferramenta para avaliação de imóveis e outros usos relacionados ao mercado imobiliário.

1.1 Objetivos

1.1.1 Objetivo Geral

Realizar uma análise e modelagem do valor de imóveis na cidade de João Pessoa, Paraíba, utilizando ferramentas computacionais e técnicas de aprendizagem de máquina, com o intuito de compreender os fatores que influenciam esses valores e propor modelos preditivos eficientes.

1.1.2 Objetivos Específicos

- Definir um modelo para a predição de imóveis da cidade de João Pessoa para ajudar na tomada de decisão de avaliação de imóveis.
- Desenvolver uma aplicação prática e interativa que permita a entrada de características dos imóveis e forneça estimativas de seus valores com base nos modelos construídos.
- Criar visualizações e relatórios que facilitem a interpretação dos resultados e apoiem a tomada de decisões no mercado imobiliário.

1.2 Organização do Trabalho

Além deste capítulo de introdução, em que é apresentado os problemas que serão abordados no trabalho e toda discussão envolvida, este trabalho também é composto por 6 outros capítulos. Os outros capítulos estão organizados da seguinte forma:

- **Capítulo 2:** No capítulo 2 serão apresentadas todas as ferramentas computacionais utilizadas, linguagens de programação, bibliotecas e sobre o processo para a coleta dos dados através da técnica de raspagem de dados.
- **Capítulo 3:** Aqui é apresentada toda a descrição teórica de cada um dos modelos e algoritmos utilizados.
- **Capítulo 4:** O capítulo 4 apresenta todo o processo para obtenção dos dados e a descrição da variável dos dados. Além disso, são apresentados também as etapas para a construção do modelo, as transformações realizadas nas variáveis, o processo para encontrar os hiperparâmetros ótimos dos modelos através de validação cruzada e técnicas de otimização. Por fim, para descrever as técnicas para análise de como os predições se comportam, são apresentadas e descritas as técnicas de Individual

Conditional Expectation, Local Interpretable model-agnostic explanations e Shapley Additive Explanations.

- **Capítulo 5:** No capítulo 5 são apresentados os resultados da análise exploratória, a tunagem dos modelos, os efeitos e importância das variáveis na predição.
- **Capítulo 6:** O capítulo 6 apresenta a conclusão do trabalho a partir da discussão dos resultados obtidos durante seu desenvolvimento.
- **Capítulo 7:** Por fim, o capítulo 7 apresenta a aplicação final desenvolvida e discute as suas funcionalidades e utilidade prática.

2 Recursos Computacionais

Nesta seção, serão apresentados os recursos computacionais utilizados no desenvolvimento deste trabalho. As ferramentas selecionadas incluem linguagens de programação amplamente conhecidas, como Python e R, e sistemas de publicação técnica, como o Quarto. Cada uma dessas tecnologias será descrita em relação ao seu papel na modelagem, coleta e manipulação de dados, bem como na criação da aplicação final. Portanto, a primeira etapa será descrever as tecnologias utilizadas para a coleta dos dados.

2.1 Recursos utilizados para a coleta de dados

Uma das principais dificuldades no trabalho com dados do mercado imobiliário é a escassez de informações disponíveis na internet, o que exige a busca por alternativas para sua obtenção. Uma solução amplamente utilizada é a extração direta de informações de sites especializados, um processo conhecido como Web Scraping (raspagem de dados). Essa abordagem não se restringe ao setor imobiliário, sendo igualmente aplicável a outras áreas, como a obtenção de dados sobre automóveis, entre outros.

O Web scraping é uma técnica utilizada para extrair informações de sites na internet, salvando-as em arquivos ou sistemas de banco de dados para realizar análise, construção de aplicações ou ter acesso a informações de difícil disponibilização. Geralmente a raspagem de dados é realizada utilizando o Hypertext Transfer Protocol (HTTP). O HTTP é o protocolo responsável por fazer toda a comunicação cliente-servidor contida na internet com base na definição de oito métodos de requisição: GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS e CONNECT. Cada método indica a ação a ser realizada no recurso especificado.

- GET: O método GET serve para requisitar uma representação do recurso especificado. Ou seja, ele serve para visualizar dados de um site.
- HEAD: O HEAD é bastante semelhante o GET, mas ele retorna apenas metadados sobre um recurso no servidor, sem que o recurso seja retornado. Ele retorna todos os cabeçalhos associados a um recurso em uma determinada URL.
- POST: O método POST envia dados para serem processados para o recurso especificado. Esses dados podem ser, por exemplo, dados de um formulário HTML.
- PUT: O PUT é bastante semelhante ao POST, ele envia os dados de forma semelhante. No entanto, caso seja necessário atualizar um usuário diversas vezes, o método PUT vai sobrescrever os dados e ficará apenas um único registro atualizado. Para o método POST, serão criados diversos registros para cada requisição realizada.
- DELETE: Exclui o recurso.

- TRACE: O método TRACE HTTP é usado para diagnóstico, depuração e solução de problemas. Ele simplesmente retorna um rastreamento de diagnóstico que registra dados do ciclo de forma que o cliente possa saber o que os servidores intermediários estão mudando em sua requisição.
- OPTIONS: O método OPTIONS retorna uma lista de quais métodos HTTP são suportados e permitidos pelo servidor.
- CONNECT: O CONNECT é usado para criar uma conexão com um recurso do lado do servidor. O alvo mais comum do método HTTP CONNECT é um servidor proxy, que um cliente deve acessar para sair da rede local.

Toda raspagem inicia com a composição de uma requisição HTTP para adquirir recursos de um site. Geralmente, essa requisição é formatada numa consulta GET ou em uma mensagem HTTP contendo uma consulta POST. Quando a requisição é recebida e processado com sucesso, o recurso requisitado é salvo site e enviado de volta em diversos formatos de arquivos como, HTML, XML, JSON ou dados multímidia. Após o download do conteúdo do site, o processo de extração continua com a reformatação e organização dos dados de forma estruturada.

Para realizar a raspagem de dados, foram utilizadas as linguagens de programação R (R CORE TEAM, 2024) e Python (VAN ROSSUM; DRAKE JR, 1995). R é uma linguagem voltada para computação científica e visualização de dados, desenvolvida pelos professores Ross Ihaka e Robert Gentleman. Ela foi criada inicialmente para ensinar introdução à estatística na Universidade de Auckland. A primeira versão do R foi lançada em 1993, e, em 1997, a linguagem tornou-se oficialmente parte do Projeto GNU. Em R, há diversos pacotes disponíveis para a realização de raspagem de dados. Neste trabalho, foi utilizado o pacote [xml2](#) (WICKHAM; HESTER; OOMS, 2023) e o [rvest](#) (WICKHAM, 2024).

O pacote [xml2](#) foi inspirado no [jQuery](#), uma biblioteca JavaScript projetada para facilitar tarefas como a manipulação de documentos HTML. Ele é especialmente útil para lidar com arquivos HTML e XML, oferecendo ferramentas eficientes para acessar, navegar e modificar os nós de documentos estruturados. Essas funcionalidades permitem, por exemplo, a extração de informações específicas de páginas web ou arquivos XML. Embora o [xml2](#) não possua suporte direto para compor requisições HTTP, ele se destaca por sua capacidade de interpretar a estrutura hierárquica de sites e organizar os dados extraídos de maneira estruturada e eficiente.

Embora o pacote [rvest](#) compartilhe algumas funcionalidades com o [xml2](#), ele oferece diversas características adicionais que tornam o processo de web scraping ainda mais simples. O [rvest](#) facilita o acesso direto a páginas web e a extração de informações com o uso de seletores CSS e XPath, permitindo a seleção precisa de elementos específicos da página. Uma funcionalidade particularmente interessante é o recurso [LiveHTML](#), que possibilita a interação direta com a página, abrindo-a em um navegador para observar, em tempo real, as operações realizadas no site. Isso torna o processo de coleta de dados mais intuitivo, uma vez que se torna possível observar as operações que se deseja realizar em tempo real.

Apesar de os pacotes mencionados anteriormente oferecerem diversas possibilidades para realizar raspagem de dados, muitas páginas possuem conteúdos dinâmicos que

só podem ser acessados por meio de outros pacotes, capazes de carregar o restante do conteúdo da página em questão. Nesse contexto, foi utilizado o pacote [selenium](#) (THORPE, 2024), que realiza requisições HTTP enquanto simula a experiência de um usuário real. Ele permite automatizar interações com a página, como clicar em botões, rolar a página automaticamente, lidar com autenticação, cookies e redirecionamentos, além de simular navegadores como Google Chrome, Firefox, entre outros. Por fim, um dos pacotes que também foi utilizado para fazer requisições HTTP, mas que não teve o mesmo objetivo do [selenium](#), foi o [httr](#) (WICKHAM, 2023).

Como mencionado anteriormente, [Python](#) também foi utilizado para realizar a raspagem de dados, embora tenha sido empregado em momentos subsequentes à utilização do [R](#). [Python](#) (VAN ROSSUM; DRAKE JR, 1995) é uma linguagem de programação criada por Guido van Rossum, que iniciou seu desenvolvimento no final dos anos 1980 como sucessora da linguagem [ABC](#). A primeira versão oficial do [Python](#) foi lançada em 1991. Trata-se de uma linguagem de alto nível, de propósito geral, que prioriza a simplicidade e a legibilidade do código, tornando-a amplamente adotada para diversas aplicações, incluindo análise de dados e automação de tarefas, além de ser amplamente utilizada para aprendizagem de máquina.

A linguagem de programação [Python](#) possui um ecossistema diversificado de bibliotecas para a realização de diversas tarefas, incluindo a raspagem de dados. Especificamente para essa finalidade, foi utilizada a biblioteca [Scrapy](#) (KOUZIS-LOUKAS, 2016), um framework voltado para web scraping e crawling que permite extrair dados de páginas da web de forma automatizada.

O [Scrapy](#) é uma ferramenta robusta e completa, com suporte nativo para diversos métodos HTTP. Ele foi projetado para ser rápido e eficiente, facilitando a criação de spiders — programas que navegam automaticamente por páginas web e extraem dados de forma estruturada. Além disso, o framework permite a exportação dos dados coletados para diversos formatos, como CSV, JSON ou bancos de dados, tornando-o uma solução ideal para projetos de coleta de informações em grande escala.

O framework também possui seletores [XPath](#) e [CSS](#), que facilitam a localização e extração de informações específicas de páginas HTML. Uma funcionalidade especialmente interessante é sua capacidade de lidar com redirecionamentos de IP, permitindo superação de restrições impostas por alguns sites durante o processo de raspagem. Além disso, o [Scrapy](#) permite a criação de [pipelines](#), que são etapas de processamento dos dados em tempo de execução. Essas [pipelines](#) são extremamente úteis para transformar, limpar e organizar as informações antes de serem salvas, garantindo que os dados extraídos já estejam em um formato estruturado e pronto para análise ou armazenamento.

As middlewares no [Scrapy](#) são componentes intermediárias que permitem modificar e processar requisições e respostas em tempo de execução. Elas são fundamentais para personalizar o comportamento do framework durante o fluxo de raspagem. Existem dois tipos principais de middlewares no [Scrapy](#): middlewares de requisição e middlewares de download.

As middlewares de requisição são utilizadas, por exemplo, para adicionar cabeçalhos personalizados às requisições ou alterar o agente do usuário ([User-Agent](#)) para evitar

bloqueios por parte dos servidores. Uma API bastante interessante que possui integração completa com o [Scrapy](#) e que foi empregada neste trabalho foi o uso de uma API gratuita oferecida pelos desenvolvedores da ScrapeOps, que permite o rotacionamento automático de cabeçalhos, como o [User-Agent](#). Essa API está disponível em <https://scrapeops.io/>.

Por outro lado, as middlewares de download têm a função de gerenciar o processo de download das páginas. Elas podem ser configuradas para evitar redirecionamentos indesejados, implementar atrasos entre requisições para evitar sobrecarregar os servidores e diminuir as chances de ser bloqueado. Essas middlewares tornam o [Scrapy](#) altamente flexível, permitindo ajustes específicos para atender às necessidades do projeto.

Embora o [Scrapy](#) ofereça uma ampla gama de funcionalidades, pode ser necessário recorrer a ferramentas externas para lidar com conteúdos dinâmicos presentes em algumas páginas web. O [Scrapy](#) possui integração com bibliotecas especializadas que facilitam o tratamento desse tipo de conteúdo. Uma dessas bibliotecas, utilizada neste trabalho, é o [scrapy-playwright](#), que integra o poder do [Playwright](#) ao [Scrapy](#).

O [Playwright](#) foi originalmente desenvolvido para realizar testes em aplicações web e executar tarefas de automação. No entanto, aqui ele foi empregado para interagir com o conteúdo dinâmico de páginas, como carregar elementos gerados por [JavaScript](#). Sua utilização neste trabalho teve o mesmo propósito que o pacote [selenium](#) desempenhou na raspagem de dados feita com [R](#), permitindo capturar dados que não seriam acessíveis diretamente via requisições HTTP convencionais. Além disso, ele também permite simular navegadores para acompanhar em tempo real o que cada requisição está alterando. Por fim, ao coletar os dados de imóveis, foi necessário realizar também a geocodificação dos endereços. Para isso, foi utilizado o pacote [tidygeocoder](#) (CAMBON *et al.*, 2021) da linguagem [R](#), que também serviu para os dados que estavam sendo coletados em [Python](#).

Assim chega ao fim descrição das tecnologias utilizadas para a obtenção dos dados. Entretanto, a linguagem [Python](#), em particular, continuou sendo empregada em outras etapas deste trabalho, como na modelagem dos valores de imóveis. Portanto, os próximos recursos apresentados serão aqueles utilizados especificamente para a modelagem e visualização dos resultados.

2.2 Recursos utilizados para modelagem e visualização dos resultados

Com os dados coletados, a etapa de modelagem dos valores de imóveis torna-se essencial para converter essas informações em entendimentos relevantes e aplicáveis. Essa etapa permite identificar padrões de comportamento entre as variáveis que influenciam os valores imobiliários, além de determinar quais fatores exercem maior impacto sobre esses valores. Para chegar a esses resultados, foram utilizadas bibliotecas desenvolvidas em [Python](#), como [scikit-learn](#) (PEDREGOSA, F. *et al.*, 2011), [pandas](#) (TEAM, 2020), empregada para a manipulação das bases de dados utilizadas neste trabalho, [numpy](#) (HARRIS *et al.*, 2020), voltada para computação numérica, entre outras.

O [scikit-learn](#) é uma das bibliotecas de aprendizado de máquina mais populares em [Python](#). Ela oferece uma extensa coleção de algoritmos para tarefas como classificação,

regressão, clusterização, além de ferramentas para pré-processamento de dados, validação cruzada e seleção de modelos. O projeto teve início no Google Summer of Code como uma iniciativa do engenheiro francês David Cournapeau. O [scikit-learn](#) foi criado como uma ferramenta baseada na biblioteca [SciPy](#) (VIRTANEN *et al.*, 2020), que é voltada para computação científica e cálculo numérico em Python.

Uma das funcionalidades mais úteis do [scikit-learn](#) é o uso de pipelines para transformar dados. As pipelines permitem combinar etapas de pré-processamento e ajuste de modelos em um único fluxo organizado. Isso não apenas simplifica o código, mas também assegura que as transformações aplicadas aos dados de treinamento sejam automaticamente replicadas nos dados de teste ou em novos dados. Além de serem úteis para tarefas mais simples, como normalização de variáveis, as pipelines permitem a inclusão de etapas personalizadas para lidar com cenários mais complexos, como tratamentos avançados de dados ou integração com ferramentas externas.

O exemplo abaixo ilustra uma pipeline para pré-processamento e modelagem. As variáveis numéricas passam por padronização, enquanto as variáveis categóricas são transformadas em variáveis binárias utilizando codificação one-hot. Por fim, os dados processados alimentam um algoritmo de Random Forest para regressão:

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestRegressor

X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    random_state=42
)

numerical_features = X.select_dtypes(include=np.number).columns
categorical_features = X.select_dtypes(include=object).columns

pipeline = Pipeline([
    ('preprocessor', ColumnTransformer([
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(), categorical_features)
    ])),
    ('model', RandomForestRegressor(
        n_estimators=100,
        random_state=42
    ))
])

pipeline.fit(X_train, y_train)
```

Outra etapa fundamental no processo de aprendizado de máquina, além do pré-processamento dos dados e do ajuste do algoritmo, é a otimização dos hiperparâmetros. O [scikit-learn](#) oferece algumas ferramentas para essa tarefa, mas elas possuem funcionalidades mais básicas e podem ser limitadas para cenários mais complexos. Dessa forma, de forma complementar, esse trabalho utilizou a biblioteca [Optuna](#) (AKIBA *et al.*, 2019), que é uma biblioteca que contém uma grande quantidade de algoritmos para otimização, como, por exemplo, métodos de otimização bayesiana.

Além das funcionalidades de otimização, o [Optuna](#) também oferece ferramentas para analisar o comportamento da função objetivo durante o processo de otimização e identificar os hiperparâmetros mais relevantes. Uma dessas ferramentas é a função `plot_param_importances`, que gera um gráfico destacando a importância relativa de cada hiperparâmetro. Além disso, como o `plot_param_importances` utiliza a biblioteca [Matplotlib](#) (HUNTER, 2007), os gráficos gerados podem ser personalizados pelo usuário.

Vale destacar que bibliotecas como o [Matplotlib](#) e o [seaborn](#) (WASKOM, 2021) desempenharam um papel fundamental na geração dos gráficos e na visualização dos dados apresentados neste trabalho. Essas ferramentas foram indispensáveis para a análise exploratória e a apresentação dos resultados de forma clara e comprehensível. Além dessas bibliotecas, também foram utilizados recursos voltados para a análise do impacto e das relações entre as variáveis e as previsões realizadas pelos modelos. Nesse contexto, destacam-se o [SHAP](#) (LUNDBERG; LEE, 2017), que fornece explicações interpretáveis para os modelos de aprendizado de máquina, e o próprio [scikit-learn](#), que foi essencial para a criação dos gráficos de ICE (Individual Conditional Expectation). Esses métodos serão detalhados na seção de metodologia deste trabalho.

Vale ressaltar que algumas bibliotecas externas, baseadas na API do [scikit-learn](#), também foram utilizadas neste trabalho. Entre elas, destacam-se a [LightGBM](#) (KE *et al.*, 2017) e a [XGBoost](#) (CHEN, T.; GUESTRIN, 2016) (CHEN, T.; GUESTRIN, 2016), empregadas para a implementação de algoritmos de aprendizado de máquina. Essas bibliotecas fornecem, respectivamente, os algoritmos de gradient boosting LightGBM e Extreme Gradient Boosting, reconhecidos por sua eficiência computacional e desempenho em tarefas de modelagem preditiva.

Conclui-se, assim, a apresentação das ferramentas empregadas na modelagem e visualização dos resultados. A seguir, serão descritas as tecnologias utilizadas no desenvolvimento da aplicação final deste trabalho. Vale destacar que o [scikit-learn](#) continuou desempenhando um papel relevante na aplicação, com sua pipeline sendo serializada no formato `.pkl` por meio da biblioteca [pickle](#). Essa biblioteca permitiu a reutilização da pipeline no back-end da aplicação, garantindo a consistência, eficiência do processamento dos dados e predição do modelo para novos dados. Segue, então, a descrição das ferramentas empregadas na construção da aplicação final.

2.3 Recursos utilizados para a criação da aplicação final

A aplicação final foi desenvolvida com o objetivo de realizar previsões de valores de imóveis e oferecer uma análise detalhada do mercado imobiliário da cidade de João Pessoa. Além de integrar a pipeline de machine learning criada durante a etapa de modelagem, a aplicação também dispõe de funcionalidades interativas que permitem aos usuários

explorar os dados e visualizar os resultados de forma clara e intuitiva. Para alcançar esses objetivos, foram empregadas tecnologias para o desenvolvimento do front-end e do back-end, que foram desenvolvidos em Python.

O front-end da aplicação foi desenvolvido utilizando a biblioteca [Dash](#) (HOSSAIN, 2019), uma ferramenta voltada para a criação de aplicativos web em Python. Desenvolvida pela [Plotly](#) (INC., 2015), a [Dash](#) permite construir interfaces gráficas completas sem a necessidade de conhecimentos avançados em desenvolvimento web, integrando tecnologias como HTML, CSS e JavaScript diretamente no ambiente Python. Entre suas principais características, destacam-se a facilidade de criar gráficos interativos, a integração com bibliotecas populares como [Plotly](#) e a capacidade de atualizar componentes de forma dinâmica por meio de funções suas funções `callback`.

Para o desenvolvimento do back-end da aplicação foi utilizado o framework [FastAPI](#), uma ferramenta moderna e eficiente para a criação de APIs em Python. O [FastAPI](#) é conhecido por sua alta performance, graças ao uso de tipagem estática e a sua facilidade de suporte assíncrono, além de permitir a criação de APIs de maneira rápida e simples. Sua integração com bibliotecas como [Pydantic](#) para validação de dados e [SQLAlchemy](#) (BAYER, 2012) para interação e conexão com bancos de dados.

No projeto, o [SQLAlchemy](#) foi utilizado para gerenciar a conexão com o banco de dados, implementado com [PostgreSQL](#), um sistema de gerenciamento de banco de dados relacional amplamente reconhecido por sua robustez e alto desempenho. Isso permitiu que os dados fossem expostos na API e consumidos no front-end. Além disso, a API foi responsável por expor a pipeline do modelo, possibilitando a realização de previsões de valores imobiliários diretamente na aplicação. Não obstante, os mapas da cidade de João Pessoa, criados com a biblioteca [Folium](#) (PYTHON-VISUALIZATION, 2020), foram servidos pela API, que expôs seus arquivos HTML para consumo no front-end.

Com as tecnologias mencionadas anteriormente e o modelo final obtido, foi possível criar uma aplicação capaz de realizar previsões para os imóveis da cidade de João Pessoa, além de proporcionar uma análise de seu setor imobiliário. No entanto, o desenvolvimento de todo o código da aplicação exigiu o uso de ferramentas que facilitassem sua escrita, desenvolvimento e organização. Assim, a seguir serão apresentadas as ferramentas utilizadas tanto para a implementação do código deste trabalho quanto para a elaboração do documento final de texto que o descreve.

2.4 Recursos utilizados para escrita de código e de documento

O desenvolvimento deste trabalho foi realizado em um computador equipado com processador AMD Ryzen 7 5800H (16 núcleos), 8 GB de memória RAM, placa de vídeo GeForce GTX 1650 e um SSD NVMe de 256 GB, operando sob o sistema Pop!_OS 22.04 LTS. Embora a máquina ofereça um desempenho geral satisfatório, a quantidade limitada de memória RAM apresentou desafios em tarefas mais intensivas, como a otimização de hiperparâmetros dos modelos. Essas tarefas frequentemente demandavam até dois dias para sua conclusão e, em alguns casos, falhavam próximo ao término devido ao alto consumo computacional. Dessa forma, a escolha das ferramentas utilizadas para a escrita do código foram as que impactassem o mínimo possível no desempenho do sistema.

O [Visual Studio Code \(VSCode\)](#) foi a principal ferramenta utilizada para a escrita do código neste trabalho. O [VSCode](#) é um editor bastante leve e oferece suporte a diversas linguagens de programação e permite integração com inúmeras tecnologias por meio de suas extensões. Entre as extensões utilizadas, destaca-se o [vscodevim](#), que incorpora os key mappings do editor de texto Vim, originalmente criado por Bram Moolenaar e lançado em 2 de novembro de 1991. Além disso, foram empregadas extensões específicas para as linguagens [Python](#) e [R](#), tendo como principal objetivo uma melhor formatação de código, execução e identificação de possíveis erros.

Para a elaboração deste documento, foi utilizado o [Quarto](#) (ALLAIRE; DERVIEUX, 2024), uma plataforma de publicação científica desenvolvida pela empresa Posit. O [Quarto](#) é uma evolução do RMarkdown e se destaca por sua capacidade de criar documentos de alta qualidade que integram texto e código. Compatível com diversas linguagens de programação, como [R](#), [Python](#), e outras, essa ferramenta é extremamente versátil para análise de dados e geração de relatórios. Com o [Quarto](#), é possível produzir relatórios, artigos, livros, apresentações e até sites. Ele é amplamente adotado na comunidade científica, especialmente entre usuários de [R](#), e oferece suporte a Markdown e [LATEX](#), o que facilita a inclusão de fórmulas matemáticas, gráficos, tabelas e outros elementos visuais. Além disso, os documentos gerados podem ser exportados para diversos formatos, como [HTML](#), [PDF](#), [MS Word](#), entre outros.

No contexto deste trabalho, o [Quarto](#) foi utilizado para a produção de todo o texto, garantindo conformidade com as normas da ABNT. Sua capacidade de integrar texto, código e gráficos de maneira organizada foi essencial para facilitar o desenvolvimento do documento.

3 Algoritmos de Aprendizado de Máquina

Neste capítulo, serão descritos os algoritmos de aprendizado de máquina utilizados neste trabalho. Alguns dos métodos utilizados podem fazer uso de diversos algoritmos ou modelos estatísticos. No entanto, o foco principal e o mais utilizado foram as árvores de decisão, especialmente em sua forma particular, as árvores de regressão. Assim, os algoritmos descritos são métodos baseados em árvores.

Os métodos baseados em árvore envolvem a estratificação ou segmentação do espaço dos preditores¹ em várias regiões simples. Dessa forma, todos os algoritmos utilizados neste trabalho partem dessa ideia. Portanto, o primeiro a ser explicado será o de árvores de decisão, pois fundamenta todos os outros algoritmos. Depois das árvores de decisão, serão explicados os métodos ensemble e, por fim, diferentes variações do método de gradient boosting.

3.1 Árvores de decisão

Árvores de decisão podem ser utilizadas tanto para regressão quanto para classificação. Elas servem de base para os modelos baseados em árvores empregados neste trabalho, focando particularmente nas árvores de regressão². O processo de construção de uma árvore se baseia no particionamento recursivo do espaço dos preditores, onde cada particionamento é chamado de nó e o resultado final é chamado de folha ou nó terminal. Em cada nó, é definida uma condição e, caso essa condição seja satisfeita, o resultado será uma das folhas desse nó. Caso contrário, o processo segue para o próximo nó e verifica a próxima condição, podendo gerar uma folha ou outro nó. Veja um exemplo na Figura 3.1.

O espaço dos preditores é dividido em J regiões distintas e disjuntas denotadas por R_1, R_2, \dots, R_J . Essas regiões são construídas em formato de caixa de forma a minimizar a soma dos quadrados dos resíduos. Dessa forma, pode-se modelar a variável resposta como uma constante c_j em cada região R_j

$$f(x) = \sum_{j=1}^J c_j I(x \in R_j)$$

O estimador para a constante c_j é encontrado pelo método de mínimos quadrados. Assim, deve-se minimizar $\sum_{x_i \in R_j} [y_i - f(x_i)]^2$. No entanto, perceba que $f(x_i)$ está sendo avaliado somente em um ponto específico x_i , o que reduzirá $f(x_i)$ para uma constante c_j . É fácil de se chegar ao resultado se for observada a definição da função indicadora

¹O espaço dos preditores é o conjunto de todos os valores possíveis para as variáveis independentes \mathbf{x}

²Uma árvore de regressão é um caso específico da árvore de decisão, mas para regressão.

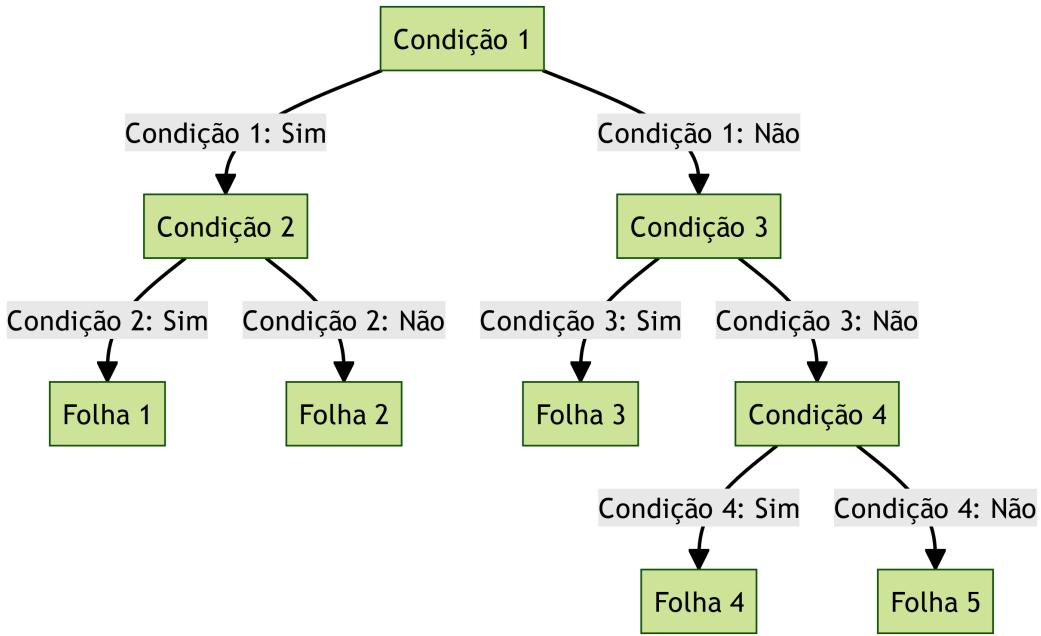


Figura 3.1: Exemplo de estrutura de árvore de regressão. A árvore tem cinco folhas e quatro nós internos.

$$I(x \in R_j)$$

$$I_{R_j}(x_i) = \begin{cases} 1, & \text{se } x_i \in R_j \\ 0, & \text{se } x_i \notin R_j \end{cases}$$

Como as regiões são disjuntas, x_i não pode estar simultaneamente em duas regiões. Assim, para um ponto específico x_i , apenas um dos casos da função indicadora será diferente de 0. Portanto, $f(x_i) = c_j$. Agora, derivando $\sum_{x_i \in R_j} (y_i - c_j)^2$ em relação a c_j

$$\frac{\partial}{\partial c_j} \sum_{x_i \in R_j} (y_i - c_j)^2 = -2 \sum_{x_i \in R_j} (y_i - c_j) \quad (3.1)$$

e igualando Equação 3.1 a 0, tem-se a seguinte igualdade

$$\sum_{x_i \in R_j} (y_i - \hat{c}_j) = 0$$

que se abrirmos o somatório e dividirmos pelo número total de pontos N_j na região R_j , teremos que o estimador de c_j será simplesmente a média dos y_i na região R_j :

$$\sum_{x_i \in R_j} y_i - \hat{c}_j N_j = 0 \Rightarrow \hat{c}_j = \frac{1}{N_j} \sum_{x_i \in R_j} y_i \quad (3.2)$$

No entanto, JAMES *et al.* (2013) caracteriza como inviável considerar todas as possíveis partições do espaço das variáveis em J caixas devido ao alto custo computacional. Dessa forma, a abordagem a ser adotada é uma divisão binária recursiva. O processo começa no topo da árvore de regressão, o ponto em que contém todas as observações, e continua

sucessivamente dividindo o espaço dos preditores. As divisões são indicadas como dois novos ramos na árvore, como pode ser visto na Figura 3.1.

Para executar a divisão binária recursiva, deve-se primeiramente selecionar a variável independente X_j e o ponto de corte s tal que a divisão do espaço dos preditores conduza a maior redução possível na soma dos quadrados dos resíduos. Dessa forma, definimos dois semi-planos

$$R_1(j, s) = \{X|X_j \leq s\} \text{ e } R_2(j, s) = \{X|X_j > s\}$$

e procuramos a divisão da variável j e o ponto de corte s que resolve a equação

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

em que c_1 e c_2 é a média da variável dependente para as observações de treinamento nas regiões $R_1(j, s)$ e $R_2(j, s)$, respectivamente. Assim, encontrando a melhor divisão, os dados são particionados nas duas regiões resultantes e o processo de divisão é repetido em todas as outras regiões.

O tamanho da árvore pode ser considerado um hiperparâmetro para regular a complexidade do modelo, pois uma árvore muito grande pode causar sobreajuste aos dados de treinamento, capturando não apenas os padrões relevantes, mas também o ruído. Como resultado, o modelo pode apresentar bom desempenho nos dados de treinamento, mas falhar ao lidar com novos dados devido à sua incapacidade de generalização. Por outro lado, uma árvore muito pequena pode não captar padrões, relações e estruturas importantes presentes nos dados. Dessa forma, a estratégia adotada para selecionar o tamanho da árvore consiste em crescer uma grande árvore T_0 , interrompendo o processo de divisão apenas ao atingir um tamanho mínimo de nós. Posteriormente, a árvore T_0 é podada utilizando o critério de custo complexidade, que será definido a seguir.

Para o processo de poda da árvore, definimos uma árvore qualquer T que pode ser obtida através do processo da poda de T_0 , de modo que $T \subset T_0$. Assim, sendo N_j a quantidade de pontos na região R_j , seja

$$Q_j(T) = \frac{1}{N_j} \sum_{x_i \in R_j} (y_i - \hat{c}_j)^2$$

uma medida de impureza do nó pelo erro quadrático médio. Assim, define-se o critério de custo complexidade

$$C_\alpha(T) = \sum_{m=1}^{|T|} N_j Q_j(T) + \alpha |T|$$

onde $|T|$ denota a quantidade total de folhas, e $\alpha \geq 0$ é um hiperparâmetro que equilibra o tamanho da árvore e a adequação aos dados. A ideia é encontrar, para cada α , a árvore $T_\alpha \subset T_0$ que minimiza $C_\alpha(T)$. Valores grandes de α resultam em árvores menores, enquanto valores menores resultam em árvores maiores, e $\alpha = 0$ resulta na própria árvore T_0 . A busca por T_α envolve colapsar sucessivamente o nó interno que provoca o menor aumento em $\sum_j N_j Q_j(T)$, continuando o processo até produzir uma árvore com um único

nó. Esse processo gera uma sequência de subárvores, na qual existe uma única subárvore menor que, para cada α , minimiza $C_\alpha(T)$.

A estimativa de α é realizada por validação cruzada com cinco ou dez folds, sendo $\hat{\alpha}$ escolhido para minimizar a soma dos quadrados dos resíduos durante o processo de validação cruzada. Assim, a árvore final será $T_{\hat{\alpha}}$. O Algoritmo 3.1 exemplifica o processo de crescimento de uma árvore de regressão:

Algoritmo 3.1 Algoritmo para crescer uma árvore de regressão

1. Use a divisão binária recursiva para crescer uma árvore grande T_0 nos dados de treinamento, parando apenas quando cada folha tiver menos do que um número mínimo de observações.
 2. Aplique o critério custo de complexidade à árvore grande T_0 para obter uma sequência de melhores subárvores T_α , em função de α .
 3. Use validação cruzada K -fold para escolher α . Isto é, divida as observações de treinamento em K folds. Para cada $k = 1, \dots, K$:
 - (a) Repita os Passos 1 e 2 em todos os folds, exceto no k -ésimo fold dos dados de treinamento.
 - (b) Avalie o erro quadrático médio de previsão nos dados no k -ésimo fold deixado de fora, em função de α . Faça a média dos resultados para cada valor de α e escolha α que minimize o erro médio.
 4. Retorne a subárvore $T_{\hat{\alpha}}$ do Passo 2 que corresponde ao valor estimado de α .
-

Algoritmo 3.1: Fonte: JAMES *et al.* (2013, p. 337).

No caso de uma árvore de decisão para classificação, a principal diferença está no critério de divisão dos nós e na poda da árvore. Para a classificação, a previsão em um nó j , correspondente a uma região R_j com N_j observações, será simplesmente a classe majoritária. Assim, tem-se

$$\hat{p}_{jk} = \frac{1}{N_j} \sum_{x_i \in R_j} I(y_i = k)$$

como a proporção de observações da classe k no nó j . Dessa forma, as observações no nó j são classificadas na classe k ($j = \arg \max_k \hat{p}_{jk}$, que é a moda no nó j).

Para a divisão dos nós no caso da regressão, foi utilizado o erro quadrático médio como medida de impureza. Para a classificação, algumas medidas comuns para $Q_j(T)$ são o erro de classificação, o índice de Gini ou a entropia cruzada.

3.2 Métodos Ensemble

As árvores de decisão são conhecidas por sua alta interpretabilidade, mas geralmente apresentam um desempenho preditivo inferior em comparação com outros modelos e algoritmos. No entanto, é possível superar essa limitação construindo um modelo preditivo

que combina a força de uma coleção de estimadores base, um processo conhecido como aprendizado em conjunto (Ensemble Learning). De acordo com HASTIE *et al.* (2009), o aprendizado em conjunto pode ser dividido em duas etapas principais: a primeira etapa consiste em desenvolver uma população de algoritmos de aprendizado base a partir dos dados de treinamento, e a segunda etapa envolve a combinação desses algoritmos para formar um estimador agregado. Portanto, nesta seção, serão definidos os métodos de aprendizado em conjunto utilizados neste trabalho.

3.2.1 Bagging

O algoritmo de Bootstrap Aggregation, ou Bagging, foi introduzido por BREIMAN (1996). Sua ideia principal é gerar um estimador agregado a partir de múltiplas versões de um preditor, que são criadas por meio de amostras bootstrap do conjunto de treinamento, utilizadas como novos conjuntos de treinamento. O Bagging pode ser empregado para melhorar a estabilidade e a precisão de modelos ou algoritmos de aprendizado de máquina, além de reduzir a variância e evitar o sobreajuste. Por exemplo, o Bagging pode ser utilizado para melhorar o desempenho da árvore de regressão descrita anteriormente.

BREIMAN (1996) define formalmente o algoritmo de Bagging, que utiliza um conjunto de treinamento \mathcal{L} . A partir desse conjunto, são geradas amostras bootstrap $\mathcal{L}^{(B)}$ com B réplicas, formando uma coleção de modelos $\{f(x, \mathcal{L}^{(B)})\}$, onde f representa um modelo estatístico ou algoritmo treinado nas amostras bootstrap para prever ou classificar uma variável dependente y com base em variáveis independentes \mathbf{x} . Se a variável dependente y for numérica, a predição é obtida pela média das previsões dos modelos:

$$f_B(x) = \frac{1}{B} \sum_{b=1}^B f(x, \mathcal{L}^{(B)})$$

onde f_B representa a predição agregada. No caso em que y prediz uma classe, utiliza-se a votação majoritária. Ou seja, se estivermos classificando em classes $j \in 1, \dots, J$, então $N_j = \#\{B; f(x, \mathcal{L}^{(b)}) = j\}$ representa o número de vezes que a classe j foi predita pelos estimadores. Assim,

$$f_B(x) = \arg \max_j N_j$$

isto é, o j para o qual N_j é máximo

Embora a técnica de Bagging possa melhorar o desempenho de uma árvore de regressão ou de classificação, isso geralmente vem ao custo de menor interpretabilidade. Quando o Bagging é aplicado a uma árvore de regressão, construímos B árvores de regressão usando B réplicas de amostras bootstrap e tomamos a média das previsões resultantes (JAMES *et al.*, 2013). Nesse processo, as árvores de regressão crescem até seu máximo, sem passar pelo processo de poda, resultando em cada árvore individual com alta variância e baixo viés. No entanto, ao agregar as previsões das B árvores, a variância é reduzida.

Para mitigar a falta de interpretabilidade do método Bagging aplicado a árvores de regressão, pode-se usar a medida de impureza baseada no erro quadrático médio, definida anteriormente, como uma métrica de importância das variáveis independentes. Um valor elevado na redução total média do erro quadrático médio, calculado com base nas divisões

realizadas por um determinado preditor em todas as B árvores, indica que o preditor é importante.

As árvores construídas pelo algoritmo de árvore de decisão se beneficiam da proposta de agregação do Bagging, mas esse benefício é limitado devido à correlação positiva existente entre as árvores. Se as árvores forem variáveis aleatórias independentes e identicamente distribuídas, cada uma com variância σ^2 , a variância da média das previsões das B árvores será $\frac{1}{B}\sigma^2$. No entanto, se as árvores forem apenas identicamente distribuídas, mas não necessariamente independentes, e apresentarem uma correlação positiva ρ , a esperança da média das B árvores será a mesma que a esperança de uma árvore individual. Portanto, o viés do agregado das árvores é o mesmo das árvores individuais, e a melhoria é alcançada apenas pela redução da variância. A variância da média das previsões será dada por:

$$\rho\sigma^2 + \frac{1 - \rho}{B}\sigma^2 \quad (3.3)$$

Isso significa que, à medida que o número de árvores B aumenta, o segundo termo da soma se torna menos significativo. Portanto, os benefícios da agregação proporcionados pelo algoritmo de Bagging são limitados pela correlação entre as árvores (HASTIE *et al.*, 2009). Mesmo com o aumento do número de árvores no Bagging, a correlação entre elas impede que as previsões individuais sejam completamente independentes, resultando em menor diminuição da variância da média das previsões do que seria esperado se as árvores fossem totalmente independentes. Uma maneira de melhorar o algoritmo de Bagging é por meio do Random Forest, que será descrito a seguir.

3.2.2 Random Forest

O algoritmo Random Forest é uma técnica derivada do método de Bagging, mas com modificações específicas na construção das árvores. O objetivo é melhorar a redução da variância ao diminuir a correlação entre as árvores, sem aumentar significativamente a variabilidade. Isso é alcançado durante o processo de crescimento das árvores por meio da seleção aleatória de variáveis independentes.

No algoritmo Random Forest, ao construir uma árvore a partir de amostras bootstrap, selecionam-se aleatoriamente $m \leq p$ das p variáveis independentes como candidatas para a divisão, antes de cada ramificação (com $m = p$ no caso do Bagging). Dessa forma, diferente do Bagging, aqui não se considera todas as p variáveis independentes para realizar a divisão e minimizar a impureza, mas apenas m dessas p variáveis. A escolha aleatória de apenas m covariáveis como candidatas para a divisão ajuda a solucionar um dos principais problemas do algoritmo de Bagging, que tende a gerar árvores de decisão semelhantes, resultando em previsões altamente correlacionadas. O Random Forest busca diminuir esse problema ao criar oportunidades para que diferentes preditores sejam considerados. Em média, uma fração $(p - m)/p$ das divisões nem sequer incluirá o preditor mais forte como candidato, permitindo que outros preditores tenham a chance de serem selecionados (JAMES *et al.*, 2013). Esse mecanismo reduz a correlação entre as árvores, o que, por sua vez, diminui a variabilidade das previsões produzidas pelas árvores.

A quantidade de variáveis independentes m selecionadas aleatoriamente é um hiperparâmetro que pode ser estimado por meio de validação cruzada. Valores comuns para m são $m = \sqrt{p}$ com tamanho mínimo do nó igual a um para classificação, e

Algoritmo 3.2 Algoritmo de uma Random Forest para regressão ou classificação

1. Para $b = 1$ até B :

- (a) Construa amostras bootstrap \mathcal{L}^* de tamanho N dos dados de treinamento.
- (b) Faça crescer uma árvore de floresta aleatória T_b para os dados bootstrap, repetindo recursivamente os seguintes passos para cada folha da árvore, até que o tamanho mínimo do nó n_{min} seja atingido.
 - i. Selecione m variáveis aleatoriamente entre as p variáveis.
 - ii. Escolha a melhor variável entre as m .
 - iii. Divida o nó em dois subnós.

2. Por fim, o conjunto de árvores $\{T_b\}_1^B$ é construído.

No caso da regressão, para fazer uma predição em um novo ponto x , temos a seguinte função:

$$\hat{f}_{rf}^B(x) = \frac{1}{B} \sum_{b=1}^B T_b(x)$$

Para a classificação é utilizado o voto majoritário. Assim, seja $\hat{C}_b(x)$ a previsão da classe da árvore de floresta aleatória b . Então,

$$\hat{C}_{rf}^B(x) = \arg \max_c \sum_{b=1}^B I(\hat{C}_b(x) = c)$$

onde c representa as classes possíveis.

Algoritmo 3.2: Fonte: HASTIE *et al.* (2009, p. 588).

$m = p/3$ com tamanho mínimo do nó igual a cinco para regressão (HASTIE *et al.*, 2009). Quando o número de variáveis é grande, mas poucas são realmente relevantes, o algoritmo Random Forest pode ter um desempenho inferior com valores pequenos de m , pois isso reduz as chances de selecionar as variáveis mais importantes. No entanto, usar um valor pequeno de m pode ser vantajoso quando há muitos preditores correlacionados. Além disso, assim como no Bagging, a Random Forest não sofre de sobreajuste com o aumento da quantidade de árvores B . Portanto, é suficiente usar um B grande o bastante para que a taxa de erro se estabilize (JAMES *et al.*, 2013).

3.2.3 Boosting Trees

O Boosting, assim como o Bagging, é um método destinado a melhorar o desempenho de modelos ou algoritmos. No entanto, neste trabalho, o Boosting foi aplicado apenas às árvores de regressão. Portanto, a explicação do Boosting será restrito ao caso de Boosting Trees (Algoritmo 3.3).

Algoritmo 3.3 Método Boosting aplicado a árvores de regressão

1. Defina $\hat{f}(x) = 0$ e $r_i = y_i$ para todos os i no conjunto de treinamento
2. Para $b = 1, 2, \dots, B$, repita:
 - (a) Ajuste uma árvore \hat{f}^b com d divisões para os dados de treinamento (X, r) .
 - (b) Atualize \hat{f} adicionando uma versão com o hiperparâmetro λ de taxa de aprendizado:

$$\hat{f}(x) \leftarrow \hat{f}(x) + \lambda \hat{f}^b(x)$$

- (c) Atualize os resíduos,

$$r_i \leftarrow r_i - \lambda \hat{f}^b(x_i)$$

3. Retorne o modelo de boosting,

$$\hat{f}(x) = \sum_{b=1}^B \lambda \hat{f}^b(x)$$

Algoritmo 3.3: Fonte: JAMES *et al.* (2013, p. 349).

No algoritmo de Bagging, cada árvore é construída e ajustada utilizando amostras bootstrap, e ao final, um estimador agregado φ_B é formado a partir das B árvores. O Boosting Trees funciona de maneira semelhante, mas sem o uso de amostras bootstrap. A ideia principal é corrigir os erros das árvores anteriores, ajustando as novas árvores aos resíduos das anteriores, visando melhorar suas previsões. Assim, as árvores são construídas de forma sequencial, incorporando as informações das árvores anteriores.

No caso da regressão, o Boosting combina um grande número de árvores de decisão $\hat{f}^1, \dots, \hat{f}^B$. A primeira árvore é construída utilizando o conjunto de dados original, e seus resíduos são calculados. Com a primeira árvore ajustada, a segunda árvore é ajustada aos da árvore anterior resíduos e, em seguida, é adicionada ao estimador para atualizar os resíduos. Dessa forma, os resíduos servem como informação crucial para construir novas árvores e corrigir os erros das árvores anteriores. Como cada nova árvore depende das árvores já construídas, árvores menores são suficientes (JAMES *et al.*, 2013).

O processo de aprendizado no método de Boosting é lenta, o que acaba gerando melhores resultados. Esse processo de aprendizado pode ser controlado por um hiperparâmetro λ chamado de shrinkage, ou taxa de aprendizado, permitindo que mais árvores, com formas diferentes, corrijam os erros das árvores passadas. No entanto, um valor muito pequeno para λ requer uma quantidade muito maior B de árvores e, diferente do Bagging e Random Forest, o Boosting pode sofrer de sobreajuste se a quantidade de árvores é muito grande. Além disso, a quantidade de divisões d em cada árvore, que controla a complexidade do boosting, pode ser considerado também um hiperparâmetro. Para $d = 1$ é ajustado um

modelo aditivo, já que cada termo envolve apenas uma variável. JAMES *et al.* (2013) define d como a profundidade de interação que controla a ondem de interação do modelo boosting, já que d divisões podem envolver no máximo d variáveis.

3.2.4 Stacked generalization

A Stacked Generalization, ou Stacking, é um método de ensemble que consiste em treinar um modelo gerado a partir da combinação da predição de vários outros modelos, visando melhorar a precisão das previsões. Esse método pode ser aplicado a qualquer modelo estatístico ou algoritmo de aprendizado de máquina. A ideia principal é atribuir pesos às previsões, de modo a dar maior importância aos modelos que produzem melhores resultados, ao mesmo tempo em que se evita atribuir altos pesos a modelos com alta complexidade.

Matematicamente, o Stacking define previsões $\hat{f}_m^{-i}(x)$ em x , utilizando o modelo m , aplicado ao conjunto de treinamento com a i -ésima observação removida (HASTIE *et al.*, 2009). Assim, os pesos são estimados de forma a minimizar o erro de predição combinado, dado pela seguinte expressão:

$$\hat{w}^{st} = \arg \min_w \sum_{i=1}^N \left[y_i - \sum_{m=1}^M w_m \hat{f}_m^{-i}(x_i) \right]^2$$

A previsão final dos modelos empilhados é $\sum_m \hat{w}_m^{st} \hat{f}_m(x)$. Assim, em vez de escolher um único modelo, o método de Stacking combina os modelos utilizando pesos estimados, o que melhora a performance preditiva, mas pode comprometer a interpretabilidade.

3.2.5 Gradient Boosting

O algoritmo de Gradient Boosting é semelhante ao de Boosting, mas com diferenças mínimas. Ele constrói modelos aditivos ajustando sequencialmente funções bases aos pseudos-resíduos, que correspondem aos gradientes da função perda do modelo atual (FRIEDMAN, 2002). Esses gradientes indicam a direção na qual a função perda diminui. Neste trabalho, foram utilizadas diferentes implementações de Gradient Boosting. No entanto, todas empregam o Gradient Boosting com árvores de regressão, com algumas modificações para a construção das árvores ou para melhorar a eficiência do algoritmo existente. Assim, o algoritmo a ser explicado será o Gradient Tree Boosting (Algoritmo 3.4).

O Gradient Boosting aplicado para árvores de regressão, tem que cada função base é uma árvore de regressão com J_m folhas. Dessa forma, cada árvore de regressão tem a forma aditiva

$$h_m(x; \{b_j, R_j\}_1^J) = \sum_{j=1}^{J_m} b_{jm} I(x \in R_{jm}) \quad (3.4)$$

em que $\{R_{jm}\}_1^{J_m}$ são as regiões disjuntas que, coletivamente, cobrem o espaço de todos os valores conjuntos das variáveis preditoras \mathbf{x} . Essas regiões são representadas pelas folhas de sua correspondente árvore. Como as regiões são disjuntas, Equação 3.4 se reduz simplesmente a $h_m(x) = b_{jm}$ para $x \in R_{jm}$. Por mínimos quadrados, b_{jm} é simplesmente

Algoritmo 3.4 Gradient Tree Boosting

1. Inicialize $f_0(x) = \arg \min_{\gamma} \sum_{i=1}^N L(y_i, \gamma)$

2. Para $m = 1$ até M :

(a) Para $i = 1, 2, \dots, N$, calcule

$$r_{im} = - \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)} \right]_{f=f_{m-1}}$$

(b) Ajuste uma árvore de regressão aos pseudo-resíduos r_{im} , obtendo regiões terminais R_{jm} , $j = 1, 2, \dots, J$.

(c) Para $j = 1, 2, \dots, J_m$, calcule

$$\gamma_{jm} = \arg \min_{\gamma} \sum_{x_i \in R_{jm}} L(y_i, f_{m-1}(x_i) + \gamma)$$

(d) Atualize $f_m(x) = f_{m-1}(x) + \lambda \sum_{j=1}^J \gamma_{jm} I(x \in R_{jm})$

3. Retorne $\hat{f}(x) = f_M(x)$

Algoritmo 3.4: Fonte: HASTIE *et al.* (2009, p. 361).

a média dos pseudo-resíduos r_{im} ,

$$\hat{b}_{jm} = \frac{1}{N_{jm}} \sum_{x_i \in R_{jm}} r_{im}$$

que dão a direção de diminuição da função perda L pela expressão do gradiente da linha 2(a). Assim, cada árvore de regressão é ajustada aos r_{im} de forma a minimizar o erro das árvores anteriores. N_{jm} denota a quantidade de pontos na região R_{jm} . Por fim, o estimador é separadamente atualizado em cada região correspondente e é expresso

$$f_m(x) = f_{m-1}(x) + \lambda \sum_{j=1}^J \gamma_{jm} I(x \in R_{jm})$$

em que γ_{jm} representa a atualização da constante ótima para cada região, baseado na função perda L , dada a aproximação $f_{m-1}(x)$. O $0 < \lambda \leq 1$, assim como no algoritmo de boosting, representa o hiperparâmetro shrinkage para controlar a taxa de aprendizado. Pequenos valores de λ necessitam maiores quantidades de iterações M para diminuir o risco de treinamento.

As outras implementações de Gradient Boosting aplicadas à árvores de decisão tem seus próprios motivos de existência. Esses motivos incluem a busca por maior eficiência computacional, adição de recursos e até mesmo maior flexibilidade. As duas outras implementações utilizadas foram o Extreme Gradient Boosting e Light Gradient Boosting.

O Extreme Gradient Boosting é uma implementação altamente eficiente do algoritmo e flexível de Gradient Boosting aplicado à árvores de decisão. Além disso, é adicionado um novo recurso, técnicas de regularização para diminuir o sobreajuste do modelo. A função objetivo agora passa a ser definida da seguinte forma:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$

em que l é uma função de perda convexa diferenciável e o segundo termo *Omega* penaliza a complexidade de cada função de árvore de decisão. O termo de regularização adicional ajuda a suavizar os pesos finais para evitar o sobreajuste. Ω é definido dado como:

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|\omega\|^2$$

onde T é a quantidade de folhas na árvore, $\|\omega\|^2$ é a soma do quadrado dos pesos associados às folhas. γ e λ são os parâmetros de regularização, em que λ penaliza os pesos das folhas e γ penaliza a quantidade de folhas nas árvores. Além disso, é fácil ver que $\frac{1}{2}\lambda\|\omega\|^2$ representa a penalização L2 (Ridge). Assim, com a regularização aplicada a função objetivo, modelos e funções preditivas serão mais comuns pois simplificarão o modelo. As implementações do algoritmo de Extreme Gradient Boosting presentes em Python utilizam a regularização L2 como padrão, mas também permitem o uso da regularização L1 (Lasso).

Assim como o Gradient Boosting, as implementações existentes de Light Gradient Boosting também permite utilizar a regularização L1 ou L2, mas não utiliza nenhuma das duas por padrão. A principal diferença do Light Gradient Boosting está na forma de crescimento das árvores, primeiro é dividido a folha com a maior redução na função de erro. Além disso, tem um grande foco na velocidade e uma melhor eficiência no uso da memória através do algoritmo de Histogram-Based. Esse algoritmo faz com que invés de avaliar cada ponto quando realiza a divisão, ele divide valores contínuos em intervalos, como um histograma, o que reduz grandemente o custo computacional.

4 Metodologia

4.1 Obtenção dos dados

Os dados foram obtidos por meio de web scraping, que é uma técnica para coleta automatizada de dados de páginas da internet. Para isso, foram utilizadas as linguagens de programação R e Python. No R, os pacotes xml2 (WICKHAM; HESTER; OOMS, 2023) e rvest (WICKHAM, 2024) foram utilizados para extrair dados de páginas estáticas de forma estruturada. No Python, as bibliotecas Scrapy (KOUZIS-LOUKAS, 2016) e Playwright, desenvolvida pela Microsoft, foram empregadas, sendo esta última essencial para a interação com componentes dinâmicos da página, possibilitando a extração de informações que exigem interações como cliques ou rolagem de página para ser gerada. Além dessas ferramentas, foram implementadas técnicas de rotacionamento de IPs e de modificação das informações do usuário que acessa o site, a fim de evitar bloqueios durante o processo de coleta de dados, garantindo assim a continuidade e eficácia da extração. A API utilizada para gerenciar a rotação de informações dos usuários que acessam o site foi desenvolvida pela empresa ScrapeOps. Essa ferramenta é disponibilizada gratuitamente e pode ser acessada através do domínio <https://scrapeops.io/> após a criação de uma conta no site.

Assim, utilizando as ferramentas e técnicas de web scraping, foram coletadas as variáveis que poderiam fazer sentido para a modelagem, priorizando aquelas com menor chance de gerar problemas durante o tratamento dos dados, como, por exemplo, ter muitos valores ausentes. Ao todo, foram extraídas 25 variáveis, das quais 10 são quantitativas e 15 qualitativas nominais, sendo 13 de caráter dicotômico. No entanto, nem todas as variáveis foram obtidas diretamente por web scraping. As coordenadas geográficas, latitude e longitude, por exemplo, foram geradas por meio da geocodificação dos endereços de cada imóvel, utilizando o pacote tidygeocoder (CAMBON *et al.*, 2021) da linguagem R. Dessa forma, tem-se as seguintes variáveis:

- Valor do imóvel: variável dependente que será modelada e constitui o principal foco de análise deste trabalho;
- Valor médio do aluguel no bairro: valor médio do aluguel dos imóveis no bairro, em m^3 ;
- Área: área total do imóvel, medida em m^2 ;
- Área média do aluguel no bairro: área média dos imóveis alugados no bairro, em m^2 ;
- Condomínio: valor mensal pago pelo condomínio do imóvel;
- IPTU: imposto cobrado sobre imóveis urbanos;
- Banheiros: quantidade de banheiros disponíveis na propriedade;

- Vagas de estacionamento: número total de vagas de estacionamento disponíveis;
- Quartos: quantidade de quartos no imóvel;
- Latitude: posição horizontal, medida em frações decimais de graus;
- Longitude: posição vertical, também medida em frações decimais de graus, assim como a latitude;
- Tipo do imóvel: sete categorias foram consideradas: apartamentos, casas, casas comerciais, casas de condomínio, casas de vila, coberturas, e lotes comerciais e de condomínio;
- Endereço: nome do endereço onde o imóvel está localizado;
- Variáveis dicotômicas: indicam a presença (1) ou ausência (0) de determinadas características no imóvel, como área de serviço, academia, elevador, espaço gourmet, piscina, playground, portaria 24 horas, quadra de esportes, salão de festas, sauna, spa e varanda gourmet.

No entanto, com base nas observações realizadas durante o estudo, nem todas as variáveis coletadas foram utilizadas na modelagem do valor dos imóveis. Algumas foram excluídas devido a uma quantidade excessiva de valores ausentes, enquanto outras se mostraram pouco significativas para explicar o valor do imóvel. Após o processo de coleta e limpeza dos dados, o banco de dados final conta com 31.782 observações.

4.2 Análise exploratória de dados

A análise exploratória de dados é uma das primeiras etapas de qualquer estudo que utiliza a estatística como ferramenta principal, pois permite identificar padrões de comportamento nos dados e descobrir relações entre as variáveis estudadas. Assim, após a coleta e organização dos dados, a primeira etapa deste estudo consistiu em uma análise descritiva. Essa análise possibilitou identificar padrões entre os diferentes tipos de imóveis e como essas características podem influenciar o seu valor. Para evidenciar esses comportamentos, foram criados gráficos e tabelas que permitiram caracterizar as relações entre as variáveis independentes e a variável dependente.

4.3 Construção do modelo

No conjunto de dados extraído, foram avaliados diferentes modelos para a previsão do valor do imóvel. O valor do imóvel foi explicado por variáveis consideradas relevantes para o estudo, como: valor médio do aluguel no bairro, área, área média do aluguel no bairro, número de banheiros, vagas de estacionamento, número de quartos, latitude, longitude, tipo de imóvel e variáveis dicotômicas obtidas durante o processo de extração. As variáveis relacionadas ao valor do condomínio e IPTU foram excluídas do modelo devido à alta quantidade de valores ausentes.

Após a seleção das variáveis, o conjunto de dados foi dividido em conjuntos de treino e teste para avaliar o desempenho dos modelos. A divisão foi realizada de forma

estratificada, utilizando a classe `sklearn.model_selection.StratifiedShuffleSplit`, da biblioteca scikit-learn. Essa classe permite dividir a base de dados de maneira aleatória, preservando a proporção das classes definidas.

A estratificação foi baseada em intervalos criados a partir da variável que representa o valor dos imóveis. Foram definidos cinco intervalos: o primeiro abrange valores entre o mínimo e 200.000; o segundo, entre 200.000 e 400.000; o terceiro, entre 400.000 e 600.000; o quarto, entre 600.000 e 800.000; e o último intervalo cobre os valores de 800.000 até o máximo da variável. Dessa forma, 20% do conjunto de dados foi reservado para o teste, enquanto os 80% restantes foram utilizados para o treinamento do modelo.

Para a aplicação das ferramentas de modelagem, foram utilizadas as bibliotecas `scikit-learn`, `lightgbm` e `xgboost`. As duas últimas foram empregadas especificamente na modelagem, enquanto a primeira também foi utilizada para criar pipelines de pré-processamento de dados, que organizam etapas sequenciais de preparação necessárias para o tratamento adequado dos dados. Assim, os quatro modelos aplicados na previsão do valor do imóvel foram: Random Forest, Gradient Boosting, LightGBM e XGBoost. Por fim, foi implementado o algoritmo de Stacking, combinando os modelos previamente construídos para melhorar a performance preditiva. No Stacking, foi utilizado como preditor final o algoritmo de Light Gradient Boosting.

4.3.1 Etapas de pré-processamento

Após a organização e limpeza dos dados, foram realizadas transformações nas variáveis para aprimorar a capacidade preditiva dos modelos em relação aos valores dos imóveis. Além disso, foi aplicado um tratamento específico para lidar com valores ausentes em algumas variáveis do conjunto de dados. Esse tratamento foi restrito às variáveis com menos de 20% de valores ausentes, incluindo as variáveis de número de banheiros, quartos, vagas, valor médio do aluguel e área média do aluguel. Por outro lado, as variáveis de condomínio e IPTU apresentaram um elevado percentual de valores ausentes, com a variável de condomínio tendo quase 60% de observações ausentes e a variável IPTU mais de 80%. Devido a essa alta proporção de valores ausentes, essas variáveis foram excluídas das análises.

O método utilizado para a imputação de valores ausentes foi o algoritmo k-nearest neighbors (KNN). Esse algoritmo estima os valores ausentes de acordo com a seguinte fórmula:

$$\hat{y} = \frac{1}{k} \sum_{x_i \in N_k(x)} y_i$$

onde $N_k(x)$ representa o conjunto de k vizinhos mais próximos de x , ou seja, os pontos x_i no conjunto de dados que estão mais próximos de x . Essa proximidade é geralmente medida pela distância Euclidiana, que é a métrica padrão utilizada pela classe `KNNImputer` da biblioteca scikit-learn para imputação de valores ausentes. No processo de imputação, foi utilizado um total de 17 vizinhos, definido pelo argumento `n_neighbors` da classe `KNNImputer`.

A transformação logarítmica $\log(1 + x)$ foi aplicada para estabilizar a variância e tornar a distribuição dos regressores mais simétrica. Para as variáveis categóricas,

utilizou-se a classe `sklearn.preprocessing.OneHotEncoder`, que converte as categorias em variáveis dicotômicas, criando uma nova coluna para cada categoria. Especificamente, a `OneHotEncoder` foi aplicada à variável que representa o tipo de imóvel, transformando cada categoria em uma variável binária.

Após a aplicação da transformação logarítmica às variáveis numéricas, essas variáveis também foram padronizadas utilizando a classe `StandardScaler`, disponível no módulo `preprocessing` da biblioteca `scikit-learn`. Essa classe ajusta os dados para que fiquem na mesma escala, padronizando-os de acordo com a fórmula:

$$z = \frac{x - \mu}{\sigma}$$

onde μ representa a média e σ o desvio padrão. Essa padronização é essencial para garantir que os modelos estatísticos e de aprendizado de máquina tratem as variáveis em escalas consistentes.

4.3.2 Validação cruzada

A técnica utilizada para otimizar e determinar os hiperparâmetros dos modelos, além de identificar os vizinhos mais próximos para a imputação de valores ausentes, foi a validação cruzada. A validação cruzada serve para estimar um erro de generalização médio da seguinte forma $Err = E [L(Y, \hat{f}(X))]$, em que L é uma função perda e \hat{f} é um estimador. Existem diversas técnicas de validação cruzada, a que foi utilizada nesse trabalho é a validação cruzada K-Fold.

A validação cruzada K-Fold é uma técnica que utiliza parte dos dados para ajustar o modelo e outra parte para testá-lo. Nessa abordagem, os dados são divididos em K folds. Em cada iteração, um desses folds é reservado para testar o modelo, enquanto os $K - 1$ folds restantes são usados para treiná-lo. O modelo é ajustado nos $K - 1$ subconjuntos e avaliado no subconjunto de teste, permitindo estimar o erro de predição. Esse processo é repetido K vezes, alternando o subconjunto de teste em cada rodada, e ao final, os K erros de predição são combinados. O erro de predição estimado pela validação cruzada é dado por:

$$CV(\hat{f}) = \frac{1}{N} \sum_{i=1}^N L(y_i, \hat{f}^{-k(i)}(x_i))$$

onde N representa o número total de observações, L é a função de perda, y_i é o valor observado, x_i é a entrada correspondente, $\hat{f}^{-k(i)}$ é o modelo ajustado sem o k -ésimo subconjunto, e $k : \{1, \dots, N\} \mapsto \{1, \dots, K\}$ indica a partição à qual a observação i foi alocada por meio de randomização.

A Figura 4.1 ilustra exatamente o caso da validação cruzada por K-Fold. As divisões do conjunto de dados para treinamento são representados pela cor azul, enquanto as laranjas são os conjuntos de validação, onde o modelo, após ser ajustado, será testado com uma função perda L .

Para realizar a validação cruzada com K-Fold nos modelos empregados neste trabalho, utilizou-se uma função e uma classe da biblioteca `scikit-learn`, ambas disponíveis no módulo `model_selection`. A função utilizada foi a `cross_val_score`, que recebe o

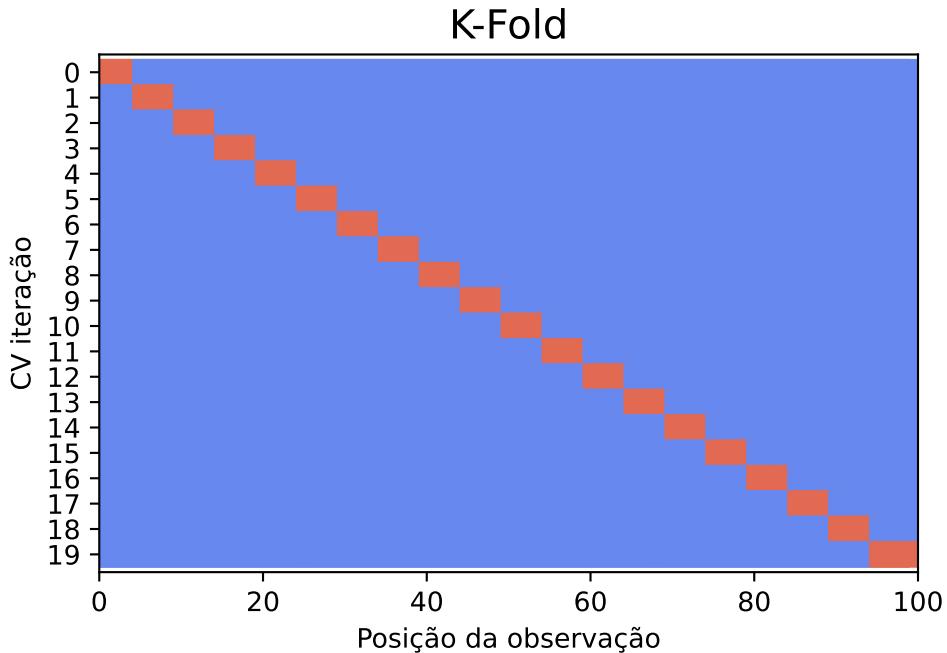


Figura 4.1: Visualização de K-Fold com 20 folds.

modelo por meio do argumento `estimator`. A função perda para avaliação é especificado pelo argumento `scoring`, enquanto a técnica de validação cruzada é definida pelo argumento `cv`. Especificamente, foi empregada a classe `KFold(n_splits=20)` para configurar a validação cruzada do tipo K-Fold, em que o parâmetro `n_splits` define o número de divisões (folds) a serem realizadas, neste caso, 20. Após a execução da validação, a média das métricas retornadas pela função `cross_val_score` para cada fold foi calculada, como forma de se obter uma estimativa do desempenho do modelo. O código abaixo é um exemplo de como se utilizar a validação cruzada em Python:

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_iris

X, y = load_iris(return_X_y=True)

model = RandomForestClassifier()

kf = KFold(n_splits=20)

scores = cross_val_score(
    estimator=model,
    X=X, y=y,
    scoring='neg_mean_squared_error',
    cv=kf)

mse_scores = - scores
mean_mse = mse_scores.mean()
```

```
print(f'Média de acurácia: {mean_mse:.4f}')
```

A métrica utilizada para avaliar o desempenho dos modelos durante a validação cruzada foi a raiz do erro quadrático médio (RMSE). O RMSE avalia, em média, o quanto os valores estimados pelo modelo se afastam dos valores observados, sendo que valores menores de RMSE indicam melhor desempenho do modelo.

Embora não tenha sido utilizada na validação cruzada, outra métrica considerada para a análise do desempenho dos modelos foi o MAPE (Erro Percentual Absoluto Médio). Essa métrica mede, em termos percentuais, o desvio médio entre os valores estimados e os valores observados, oferecendo uma interpretação relativa ao erro. Por fim, foi analisado o coeficiente de determinação (R^2), que indica a proporção da variância da variável dependente explicada pelas variáveis independentes. Neste caso, o R^2 avalia o quão bem as variáveis utilizadas para a construção do modelo conseguem representar a variação no valor do imóvel. As métricas utilizadas são definidas da seguinte forma:

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=0}^n (y_i - \hat{y}_i)^2} \quad R^2 = 1 - \frac{SS_{\text{resíduos}}}{SS_{\text{total}}} \quad \text{MAPE} = \frac{1}{n} \sum_{i=0}^n \left| 1 - \frac{y_i}{\hat{y}_i} \right|$$

em que $SS_{\text{resíduos}}$ e SS_{total} representam, respectivamente, a soma dos quadrados dos resíduos e a soma dos quadrados totais.

4.4 Otimização de hiperparâmetros

Existem diversas técnicas para a otimização de hiperparâmetros em aprendizado de máquina. Uma das mais comuns é o Grid Search. Segundo BISCHL *et al.* (2023), o Grid Search divide o intervalo contínuo de valores possíveis de cada hiperparâmetro em um conjunto de valores discretos, avaliando exaustivamente o desempenho do algoritmo para todas as combinações possíveis. No entanto, como o número de combinações cresce exponencialmente com o aumento do número de hiperparâmetros, o Grid Search apresenta um custo computacional elevado.

Por essa razão, métodos de otimização mais avançados, como a otimização bayesiana, têm ganhado destaque, pois oferecem um desempenho superior ao explorar o espaço de hiperparâmetros de maneira mais eficiente. Neste trabalho, utilizamos a otimização bayesiana para realizar a otimização dos modelos e determinar a quantidade ideal de vizinhos mais próximos na classe `KNNImputer`, empregada para a imputação de valores ausentes.

A otimização bayesiana não se refere a um algoritmo específico, mas sim a uma abordagem de otimização fundamentada na inferência bayesiana, que engloba uma ampla família de algoritmos (GARNETT, 2023). Além disso, a otimização bayesiana tem alcançado benchmarks superiores em comparação com outros algoritmos em diversos problemas complexos de otimização de hiperparâmetros (SNOEK; LAROCHELLE; ADAMS, 2012).

Diferentemente de outros algoritmos de otimização de hiperparâmetros, a otimização bayesiana ajusta suas tentativas futuras com base nos resultados obtidos anteriormente (YANG; SHAMI, 2020). Para definir os pontos de avaliação futuros, utiliza-se uma função probabilística $P(c|\lambda)$, que modela a relação entre os hiperparâmetros λ e a métrica de desempenho c (BERGSTRA; YAMINS; COX, 2013). A partir dessa função, estima-se, para cada conjunto de hiperparâmetros λ , a performance esperada $\hat{c}(\lambda)$ e a incerteza associada à predição $\hat{\sigma}(\lambda)$. Com essas estimativas, a distribuição preditiva derivada da função probabilística permite a aplicação de uma função de aquisição. Essa função orienta a escolha dos próximos pontos a serem avaliados, equilibrando exploitation (explorar regiões próximas às melhores observações anteriores) e exploration (investigar áreas ainda não exploradas)¹.

Portanto, os algoritmos de otimização bayesiana são regidos pela relação $\lambda \rightarrow c(\lambda)$ e buscam equilibrar exploitation e exploration. Isso permite identificar as regiões mais promissoras no espaço de hiperparâmetros, ao mesmo tempo em que evita negligenciar possíveis configurações melhores em áreas ainda inexploradas.

4.4.1 Tree-Structured Parzen Estimator

A função probabilística utilizada para a otimização bayesiana neste trabalho foi a Tree-Structured Parzen Estimator (TPE). O TPE define duas funções de densidade, $l(\lambda)$ e $g(\lambda)$, que são empregadas para modelar a distribuição das variáveis no domínio dos hiperparâmetros λ (YANG; SHAMI, 2020). Essas densidades são utilizadas para estimar a probabilidade condicional $p(\lambda|y)$ de observar uma combinação de hiperparâmetros λ , dado um valor de métrica de desempenho y . A definição é dada por:

$$P(\lambda|y) = \begin{cases} l(\lambda) & \text{if } y < y^* \\ g(\lambda) & \text{if } y \geq y^* \end{cases} \quad (4.1)$$

em que $l(\lambda)$ representa a densidade associada aos valores de y menores que o limiar y^* e $g(\lambda)$ é a densidade associada aos valores de y iguais ou superiores a y^* (BERGSTRA *et al.*, 2011). No algoritmo de TPE, o valor de y^* é definido como sendo um quantil γ dos valores observados de y , de forma que $p(y < y^*) = \gamma$.

Por padrão, o Tree-Structured Parzen Estimator (TPE) utiliza como função de aquisição o Expected Improvement (EI). O EI representa a expectativa de um modelo M , que mapeia $f : \Lambda \rightarrow \mathbb{R}^N$, sobre a melhora esperada em relação a um limiar y^* . Formalmente, o EI é definido como:

$$EI_{y^*}(\lambda) = \int_{-\infty}^{\infty} \max(y^* - y, 0) p_M(y|\lambda) dy$$

Se, para o valor de λ , o modelo prevê um y tal que $y > y^*$, a diferença $y^* - y$ será negativa, e o retorno será 0, o que significa que não haverá melhora. Por outro lado, se $y < y^*$, a diferença será positiva, indicando que o modelo apresenta um desempenho superior em relação ao limiar y^* .

Portanto, para o cálculo do EI utilizando as definições fornecidas pelo TPE e assumindo

¹Exploitation refere-se à busca por soluções promissoras com base em dados prévios, enquanto exploration visa descobrir novas regiões potencialmente vantajosas

que $y^* > y$, adota-se a parametrização de $p(\lambda, y)$ como $p(y)p(\lambda | y)$, com o objetivo de simplificar os cálculos. Assim, a expressão do EI, para o TPE, se reduz a:

$$EI_{y^*}(\lambda) = \int_{-\infty}^{y^*} (y^* - y) p(y|\lambda) dy = \int_{-\infty}^{y^*} (y^* - y) \frac{p(\lambda|y)p(y)}{p(\lambda)} dy \quad (4.2)$$

A partir da Equação 4.1 e da definição $p(y < y^*) = \gamma$ e $p(y \geq y^*) = 1 - \gamma$, pode-se encontrar a distribuição marginal $p(\lambda)$. Dessa forma, segue-se que:

$$\begin{aligned} p(\lambda) &= \int_{-\infty}^{\infty} p(\lambda, y) dy \\ &= \int_{-\infty}^{\infty} p(\lambda|y)p(y) dy \\ &= \int_{-\infty}^{y^*} p(\lambda|y)p(y) dy + \int_{y^*}^{\infty} p(\lambda|y)p(y) dy \\ &= \int_{-\infty}^{y^*} l(\lambda)p(y) dy + \int_{y^*}^{\infty} g(\lambda)p(y) dy \\ &= \gamma l(\lambda) + (1 - \gamma)g(\lambda) \end{aligned}$$

Agora, basta calcular a integral $\int_{-\infty}^{y^*} (y^* - y) p(\lambda|y)p(y) dy$. Tem-se, portanto:

$$\begin{aligned} \int_{-\infty}^{y^*} (y^* - y) p(\lambda|y)p(y) dy &= l(\lambda) \int_{-\infty}^{y^*} (y^* - y) p(y) dy \\ &= \gamma y^* l(\lambda) - l(\lambda) \int_{-\infty}^{y^*} y p(y) dy \end{aligned}$$

Finalmente, substituindo essa última expressão encontrada e $p(\lambda)$ na Equação 4.2, chega-se a:

$$EI_{y^*}(\lambda) = \frac{\gamma y^* l(\lambda) - l(\lambda) \int_{-\infty}^{y^*} y p(y) dy}{\gamma l(\lambda) + (1 - \gamma)g(\lambda)} \propto \left[\gamma + \frac{g(\lambda)}{l(\lambda)} (1 - \gamma) \right]^{-1}$$

Essa última expressão mostra que, para maximizar o Expected Improvement, é necessário encontrar valores de λ que apresentem alta probabilidade em $l(\lambda)$ e baixa probabilidade em $g(\lambda)$. Portanto, no TPE, maximizar o EI equivale a maximizar a razão $l(\lambda)/g(\lambda)$.

4.4.2 Otimização de hiperparâmetros com optuna

Para otimizar os hiperparâmetros dos modelos foi utilizado a bilbioteca `optuna` (AKIBA *et al.*, 2019) da linguagem de programação Python. Essa biblioteca implementa diversos métodos para otimização automatizada de hiperparâmetros. Para a sua utilização, é preciso definir inicialmente uma função objetivo. Por exemplo, para otimizar os hiperparâmetros de uma random forest, seria necessário definir uma função objetivo da seguinte forma:

```

import optuna
import numpy as np
import pandas as pd
from sklearn import ensemble
from sklearn.model_selection import cross_val_score, KFold

def objective(trial):
    X = train_df[variaveis_independentes]
    y = train_df.variavel_dependente

    params = dict(
        n_estimators=trial.suggest_int(
            name='n_estimators',
            low=1,
            high=1000),
        max_depth=trial.suggest_int(
            name='max_depth',
            low=20,
            high=1000),
        max_features='sqrt',
        random_state=42
    )

    model = ensemble.RandomForestRegressor(
        *params
    )
    model.fit(X=X, y=y)

    cv_scores = np.expm1(np.sqrt(-cross_val_score(
        estimator=model,
        X=X,
        y=y,
        scoring="neg_mean_squared_error",
        n_jobs=3,
        cv=KFold(n_splits=20)))) 

    return np.mean(cv_scores)

study = optuna.create_study()
study.optimize(objective, n_trials=100, n_jobs=-1)

```

Primeiro, define-se, em cada tentativa (trial), quais hiperparâmetros serão otimizados, especificados no objeto `params` no início da função. Após definir o espaço de busca para cada hiperparâmetro, o modelo escolhido é ajustado aos dados de treinamento. Com o modelo ajustado, realiza-se a validação cruzada em cada trial, utilizando o método K-Fold com 20 divisões (folds), conforme definido previamente. Após definir a função objetivo, inicializa-se um estudo com `optuna.create_study` e, em seguida, inicia-se a otimização com `study.optimize(objective, n_trials=100, n_jobs=-1)`. Por fim,

para selecionar os melhores hiperparâmetros ao fim do último trial, basta executar `study.best_params`.

Por padrão, a biblioteca Optuna utiliza o Tree-Structured Parzen Estimator (TPE) para otimizar hiperparâmetros de um modelo. A técnica de otimização é escolhida por meio do argumento `sampler` no método `create_study`. Para selecionar o TPE, basta passar `optuna.samplers.TPESampler` como argumento para a criação do estudo. O método TPE é o padrão para otimização na biblioteca optuna. No entanto, caso se deseje utilizar outro método de otimização da biblioteca, basta especificá-lo da mesma forma: `optuna.create_study(sampler=metodo_otimizacao)`.

4.5 Interpretação dos algoritmos de aprendizagem de máquina

Na aplicação de aprendizado de máquina, o foco geralmente está em obter um modelo com o menor erro de generalização possível, o que muitas vezes resulta na negligência da interpretação dos resultados e do que mais influenciou a variável dependente. Isso pode comprometer a compreensão do que o algoritmo está efetivamente fazendo. Em resposta a essa limitação, diversas técnicas têm sido desenvolvidas para interpretar os efeitos das variáveis independentes nas estimativas geradas pelos algoritmos. Assim, esta seção será dedicada a descrever a fundamentação teórica e a aplicação das técnicas de interpretação utilizadas neste trabalho.

4.5.1 Individual Conditional Expectation (ICE)

O método Individual Conditional Expectation (ICE) é uma ferramenta gráfica que permite visualizar as estimativas de um modelo de forma detalhada. Esse método traça a relação entre os valores preditos pelo modelo e uma variável específica, analisando cada observação individualmente. Dessa forma, o ICE possibilita a análise da variação dos valores ajustados ao longo do domínio de uma covariável, destacando tanto a heterogeneidade entre as observações quanto a forma como cada uma responde individualmente às mudanças na variável em questão. Essa abordagem facilita a identificação de padrões e variações que poderiam ser obscurecidos em análises agregadas (GOLDSTEIN *et al.*, 2015).

Formalmente, o método ICE considera as observações $\{x_{S_i}, \mathbf{x}_{C_i}\}_{i=1}^N$ e os valores preditos \hat{f} . Para cada uma das N observações, é traçada uma curva $\hat{f}_S^{(i)}$ em função dos valores da variável de interesse x_S , enquanto as demais variáveis x_C permanecem fixas em seus valores observados. Nesse gráfico, x_S é representado no eixo das abscissas, e as previsões correspondentes pelo modelo aparecem no eixo das ordenadas.

Quando há um grande número de curvas no gráfico ICE, a interpretação pode se tornar desafiadora devido à sobreposição ou excesso de linhas. Para simplificar a análise, pode-se aplicar uma técnica de centralização, conhecida como c-ICE (centered ICE). Essa abordagem centraliza as curvas em um ponto específico de x_S , mostrando apenas a diferença nas previsões em relação a esse ponto. Assim, as novas curvas c-ICE são definidas da seguinte forma:

$$\hat{f}_{cent}^{(i)} = \hat{f}^{(i)} - \mathbf{1}\hat{f}(x^*, \mathbf{x}_{C_i})$$

onde x^* é selecionado como o mínimo ou o máximo de x_S , \hat{f} é o modelo ajustado, e $\mathbf{1}$ é um vetor de uns.

Quando x^* é escolhido como o valor mínimo de x_S , todas as curvas c-ICE iniciam em zero, eliminando as diferenças de nível causadas pelas variações nos valores de $x_C^{(i)}$ entre as observações. Por outro lado, se x^* for o valor máximo de x_S , as curvas centralizadas mostram o efeito cumulativo de x_S sobre \hat{f} em relação ao ponto de centralização. Essa abordagem facilita a interpretação do impacto de x_S .

O método Individual Conditional Expectation (ICE) é similar ao Partial Dependence Plot (PDP), mas com a diferença de que o PDP apresenta uma visão agregada (global) dos efeitos das variáveis independentes nas previsões e, por outro lado, o ICE fornece uma análise mais detalhada, mostrando uma linha para cada instância do conjunto de dados. O PDP é obtido como a média das linhas geradas pelo método ICE, oferecendo uma visão mais generalizada do impacto de uma variável. Matematicamente, o PDP é definido como:

$$\hat{f}_S(x_S) = E_{\mathbf{X}_C} [\hat{f}(x_S, \mathbf{X}_C)]$$

em que \mathbf{X}_C representa o conjunto das variáveis independentes que são mantidas fixas durante a análise e x_S é a variável independente de interesse, cujo efeito sobre a previsão se deseja analisar.

Para calcular $\hat{f}_S(x_S)$ na prática, pode-se utilizar simulação de Monte Carlo. O PDP pode, então, ser estimado da seguinte forma:

$$\hat{f}_S(x_S) = \frac{1}{N} \sum_{i=1}^N \hat{f}(x_S, \mathbf{X}_{C_i})$$

onde N é o número de amostras utilizadas na simulação e $\hat{f}(x_S, \mathbf{X}_{C_i})$ são as previsões do modelo.

Em Python, a biblioteca `scikit-learn` disponibiliza a classe `PartialDependenceDisplay` para a criação de gráficos de ICE e a inclusão opcional da linha de PDP. Para gerar o gráfico, é necessário ter um modelo previamente ajustado. O código a seguir demonstra sua utilização:

```
from sklearn.inspection import PartialDependenceDisplay

PartialDependenceDisplay\
    .from_estimator(
        model,
        df,
        features,
        kind="both",
        centered=True,
        random_state=set_seed
    )
```

No exemplo acima, o método `.from_estimator` é usado para criar o gráfico diretamente

a partir de um modelo ajustado. Ele recebe como argumentos o modelo (`model`), a base de dados (`df`) e as variáveis de interesse (`features`). O argumento `kind` permite definir o tipo de visualização, podendo exibir apenas o PDP, as linhas do ICE ou ambos (`kind="both"`). Já o argumento `centered` oferece a opção de centralizar as curvas.

4.5.2 Local interpretable model-agnostic explanations (LIME)

RIBEIRO, M. T.; SINGH; GUESTRIN (2016) definem o Local Interpretable Model-Agnostic Explanations (LIME) como um algoritmo capaz de explicar as previsões realizadas por qualquer modelo de classificação ou regressão, aproximando-o localmente de um modelo mais interpretável. O objetivo principal do LIME é identificar um modelo interpretável e gerar uma representação comprehensível que permita traduzir o comportamento de modelos complexos.

Formalmente, para a construção da explicação gerada pelo LIME, a explicação é definida como um modelo $g \in G$, onde G representa uma classe de potenciais modelos interpretáveis, como regressão linear ou árvores de decisão. O domínio de g é dado por $0, 1^d$, ou seja, o modelo explicativo g opera sobre a presença ou ausência de componentes interpretáveis. Como nem todos os modelos $g \in G$ podem ser simples o suficiente para serem interpretados, define-se uma medida de complexidade $\Omega(g)$ para a explicação gerada por $g \in G$. Por exemplo, para uma árvore de regressão, $\Omega(g)$ pode ser a profundidade da árvore, enquanto que, para modelos lineares, $\Omega(g)$ pode ser o número de pesos diferentes de zero.

Seja $f : \mathbb{R}^d \rightarrow \mathbb{R}$ o modelo que está sendo explicado. Define-se, então, uma medida de aproximação $\pi_x(z)$ entre uma instância z e x , com o objetivo de estabelecer a localidade ao redor de x . Por fim, seja $L(f, g, \pi_x)$ uma estatística que quantifica o erro de g ao tentar aproximar f na localidade definida por π_x . Para garantir a interpretabilidade e a fidelidade local, é necessário minimizar $L(f, g, \pi_x)$, mantendo $\Omega(g)$ suficientemente baixo. Dessa forma, obtém-se a explicação gerada pelo LIME:

$$\xi(x) = \arg \min_{g \in G} L(f, g, \pi_x) + \Omega(g)$$

Podem existir diversas variações para L e Ω com diferentes famílias de modelos explicativos G . Uma escolha para L , por exemplo, é o erro quadrático médio.

Embora o LIME não tenha sido utilizado neste trabalho, ele é um dos métodos fundamentais para a abordagem que será apresentada a seguir, o Shapley Additive Explanations (SHAP). Em Python, o LIME está disponível por meio da biblioteca `lime`, cuja documentação pode ser acessada em <https://lime-ml.readthedocs.io/en/latest/lime.html>.

4.5.3 Shapley Additive Explanations (SHAP)

O SHapley Additive exPlanations (SHAP) é um método cujo objetivo é explicar as predições individuais de uma instância xx por meio da computação da contribuição de cada variável para o resultado da predição. Esse método é fundamentado nos valores de Shapley, que serão definidos a seguir.

Os valores de Shapley foram introduzidos por SHAPLEY (1953) e baseiam-se nos conceitos da teoria dos jogos de coalizão. Essa teoria foi adaptada para explicar previsões realizadas por modelos, representando a contribuição média de uma variável para a previsão, considerando todas as possíveis coalizões. Nesse contexto, coalizões referem-se a diferentes combinações de variáveis. Matematicamente, os valores de Shapley são definidos como:

$$\phi_i(x) = \sum_{Q \subseteq S \setminus \{i\}} \frac{|Q|! (|S| - |Q| - 1)!}{|S|!} (\Delta_{Q \cup \{i\}}(x) - \Delta_Q(x)) \quad (4.3)$$

onde Q é um subconjunto das covariáveis consideradas no modelo e S representa o conjunto completo de todas as covariáveis. A diferença $\Delta_{Q \cup \{i\}}(x) - \Delta_Q(x)$ corresponde à contribuição marginal da variável i ao ser adicionada ao subconjunto Q .

No entanto, a Equação 4.3 apresenta crescimento exponencial em termos de complexidade computacional à medida que o número de variáveis aumenta. Para contornar esse problema e reduzir o custo computacional, os valores de Shapley podem ser estimados de forma aproximada e eficiente utilizando o método de Monte Carlo, conforme a seguinte expressão:

$$\hat{\phi}_i(x) = \frac{1}{n!} \sum_{O \in \pi(n)} (\Delta_{Pre^i(O) \cup \{i\}} - \Delta_{Pre^i(O)}), \quad i = 1, \dots, n$$

em que $\pi(n)$ é o conjunto de todas as permutações ordenadas dos índices das variáveis $\{1, 2, \dots, n\}$ e $Pre^i(O)$ representa o conjunto de índices das variáveis que precedem i na permutação $O \in \pi(n)$ (ŠTRUMBELJ; KONONENKO, 2014).

O SHAP estabelece uma conexão entre os valores de Shapley e o método LIME, previamente definido. O modelo explicativo do SHAP, denotado por g , é uma função linear de variáveis binárias, construído a partir dos valores de Shapley, conforme define a expressão a seguir:

$$g(z') = \phi_0 + \sum_{j=1}^M \phi_j z'_j$$

onde g é o modelo explicativo, $z' \in \{0, 1\}^M$ é o vetor de coalizão, representando a presença ($z'_j = 1$) ou ausência ($z'_j = 0$) de cada covariável e ϕ_j denota os valores Shapley.

A partir do método SHAP, é possível ter diversas visualizações que ajudam a entender como as previsões do modelo se comportam. Neste trabalho foi utilizado o gráfico de importância das variáveis, resumo dos valores shapley e o de dependência. Os métodos serão descritos a seguir tendo como referência o livro de MOLNAR (2020).

O gráfico de importância das variáveis é bastante simples, variáveis com valores absolutos elevados de Shapley são importantes. No entanto, como se deseja obter a importância global, calcula-se a média dos valores absolutos de Shapley por variável em todo o conjunto de dados. Assim, tem-se a seguinte expressão:

$$I_j = \frac{1}{n} \sum_{i=1}^n |\phi_j^{(i)}|$$

O gráfico de resumo combina a importância das variáveis com seus efeitos, representados pelos valores de Shapley e pela variação das observações de cada variável. No eixo x estão os valores de Shapley, que tem sua variação representada por pontos, enquanto no eixo y encontram-se as variáveis, ordenadas de forma decrescente com base em sua importância. Os pontos, que representam os valores de Shapley, são coloridos de acordo com os valores altos ou baixos das observações originais de cada variável. Essa representação facilita a compreensão de como as previsões do modelo estão sendo influenciadas por cada variável, permitindo uma análise mais detalhada de seus efeitos.

Por fim, o gráfico de dependência é o mais simples de todos. Nele, os valores de Shapley de uma variável são plotados em função de suas respectivas observações. Matematicamente, é definido como:

$$\{(x_j^{(i)}, \phi_j^{(i)})\}_{i=1}^n$$

Esse gráfico permite visualizar diretamente como as observações de uma variável estão relacionadas aos seus efeitos no modelo, representados pelos valores de Shapley. Ele foi utilizado somente para analisar como as variáveis binárias se comportam.

Para aplicar o método SHAP em Python, foi utilizada a biblioteca shap. Essa biblioteca calcula os valores de Shapley com base no modelo ajustado e no algoritmo de explicação escolhido, implementado na classe `shap.Explainer`. Esse algoritmo estima os valores de Shapley de maneira eficiente e adaptada ao modelo em análise. Após obter os valores de Shapley, é possível criar gráficos como os de resumo, dependência e importância. O gráfico de dependência pode ser gerado utilizando a função `shap.dependence_plot`, enquanto os gráficos de importância e resumo são criados com a função `shap.summary_plot`. Abaixo, é apresentado um exemplo de código que utiliza a biblioteca shap para gerar esses gráficos baseado no algoritmo Stacking:

```
import shap

X1000 = shap.utils.sample(train_df, 1000)
explainer_stacking = shap.Explainer(
    model=stacking.predict,
    mask=X1000
)
shap_values_stacking = explainer_stacking(test_df)

shap.summary_plot(
    shap_values_stacking,
    test_df,
)

shap.summary_plot(
    shap_values_stacking,
    test_df,
    plot_type="bar",
)
```

```
shap.dependence_plot(  
    variavel,  
    shap_values_stacking.values,  
    test_df.values,  
    interaction_index=None,  
    )
```

Por padrão, a classe `shap.Explainer` utiliza o algoritmo de explicação considerado a melhor escolha com base no modelo passado como argumento. Neste trabalho, foi selecionado o algoritmo de explicação utilizando a classe `Permutation`. Essa escolha foi realizada automaticamente pela classe `shap.Explainer(algorithm="auto")`. O algoritmo `Permutation` funciona iterando sobre todas as permutações possíveis das variáveis, tanto na ordem original quanto na ordem inversa. Em relação ao gráfico de importância, ele é gerado de maneira semelhante ao gráfico de resumo, com a diferença de que o argumento `plot_type="bar"` é utilizado para criar o gráfico de importância. Por fim, o método `shap.utils.sample` é utilizado para selecionar aleatoriamente observações para a construção de uma amostra, a fim de estimar os valores de Shapley.

5 Resultados

5.1 Análise exploratória de dados

A análise exploratória dos dados foi realizada após a divisão entre os conjuntos de treinamento e teste. Essa abordagem foi adotada para evitar o sobreajuste do modelo e garantir que o algoritmo não aprenda com informações indisponíveis no conjunto de teste. Assim, a descritiva dos dados foi realizada utilizando o conjunto de treinamento.

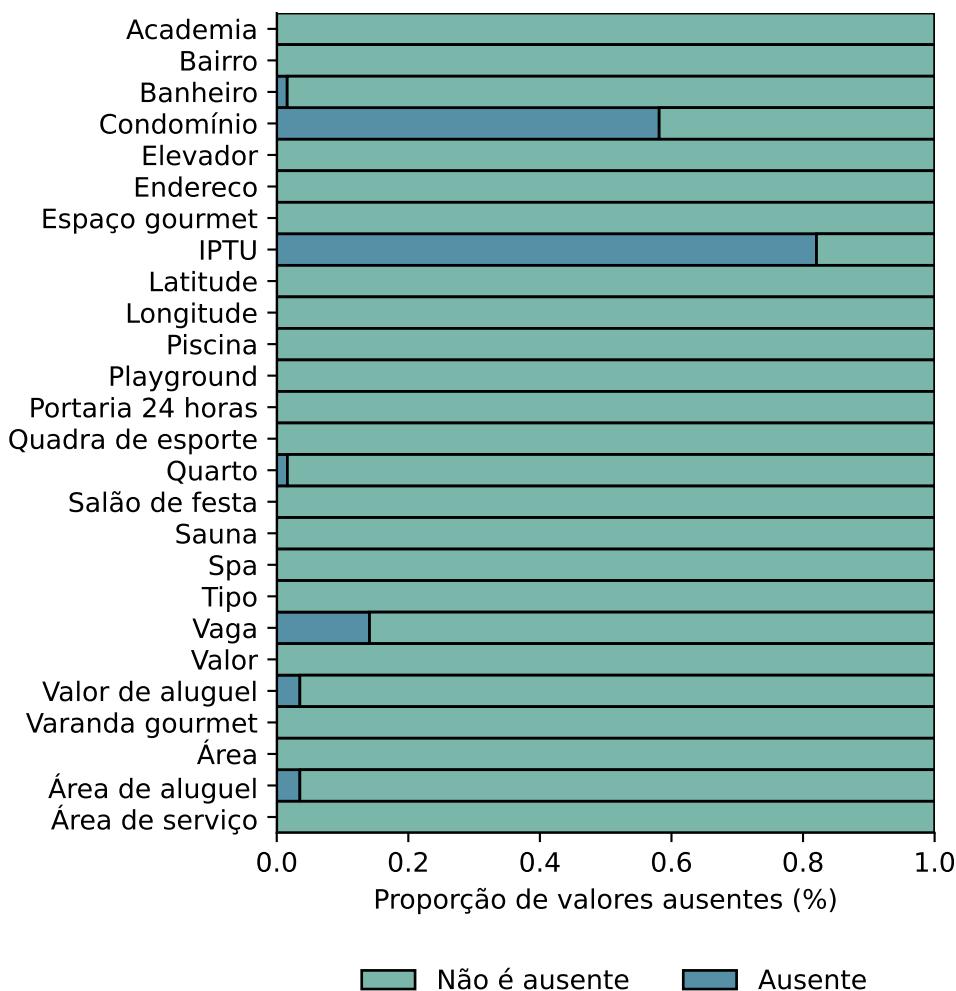


Figura 5.1: Proporção de valores ausentes por variáveis

A primeira etapa da análise exploratória de dados foi identificar os dados faltantes e determinar a melhor forma de tratá-los. A Figura 5.1 mostra a porcentagem de observações ausentes em cada variável. As variáveis com a maior quantidade de dados ausentes são o valor do condomínio e o IPTU, pois essas informações são as menos preenchidas no site de onde os dados foram coletados. A terceira variável, com quase

20% de observações ausentes, é a quantidade de vagas de estacionamento. As variáveis com mais de 20% de observações ausentes foram removidas da base de dados, pois, com essa quantidade de valores faltantes, nem mesmo métodos de imputação proporcionariam um tratamento adequado. Dessa forma, apenas as variáveis de valor do condomínio e IPTU foram removidas, enquanto as demais com valores ausentes foram tratadas por meio de imputação.

Uma das dificuldades que podem surgir durante a modelagem é o desbalanceamento das classes, ou seja, a diferença na quantidade de cada tipo de imóvel. O tipo de imóvel mais predominante no conjunto de dados são os apartamentos, que representam 81,36% do total. Em seguida, vêm as casas, com 8,91%, e os flats, com 5,72%. Por fim, as casas comerciais são as menos representadas, com apenas 15 ocorrências. Esse desbalanceamento claro entre as classes pode dificultar o desempenho do modelo, especialmente na previsão de categorias menos frequentes, como as casas comerciais, onde o modelo pode ter dificuldade em obter bons resultados.

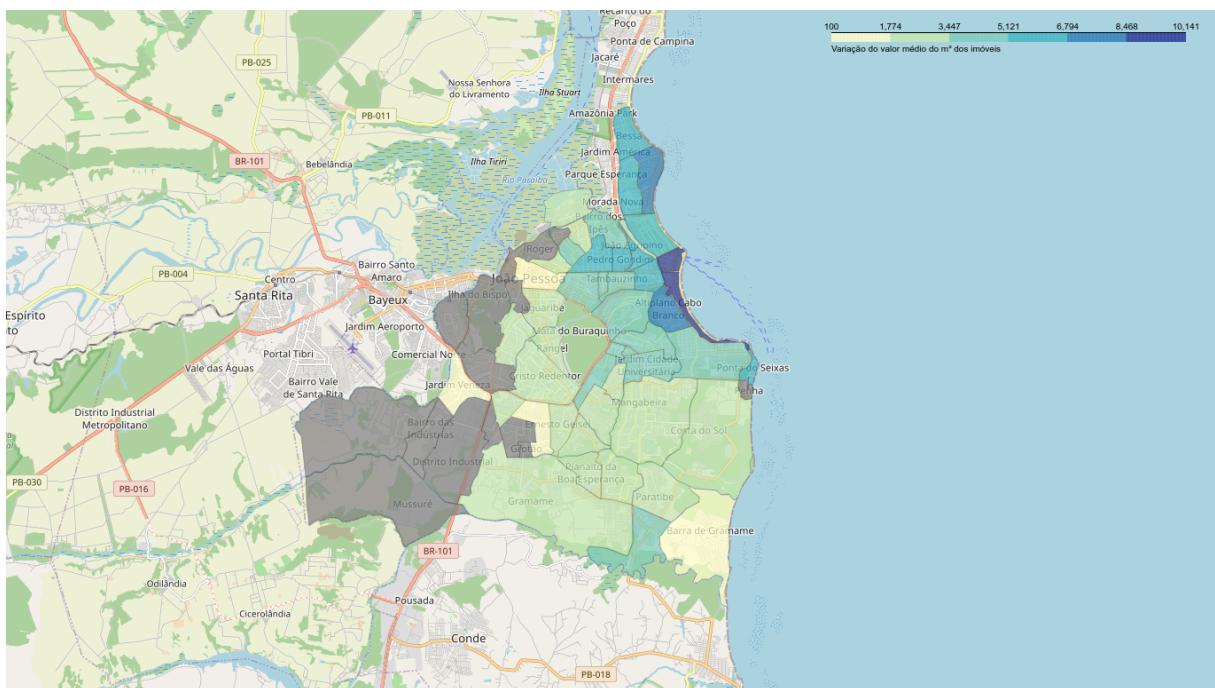


Figura 5.2: Variação da média do valor do m^2 dos imóveis de João Pessoa, área de estudo.

A partir da Figura 5.2, tem-se o mapa da região de estudo, correspondente à cidade de João Pessoa. O mapa apresenta a variação da média do valor do m^2 dos imóveis, calculada com base nos bairros da cidade. Vale destacar que alguns bairros não possuíam dados disponíveis no momento da coleta de informações por raspagem dos sites de imóveis. Esses bairros estão representados pela cor cinza. Por outro lado, os bairros com dados disponíveis apresentam variações de cores que indicam diferentes faixas de valores do m^2 . Tonalidades mais escuras representam bairros com valores médios mais altos para o m^2 , enquanto tonalidades mais claras indicam valores médios menores.

O bairro com o maior preço médio do m^2 é Cabo Branco, com um valor pouco superior a R\$ 10.000,00. Em segundo lugar, está o bairro de Tambaú, com um valor médio de R\$ 8.951,45 por m^2 . Em terceiro, encontra-se o bairro Jardim Oceanía, com um valor médio

de R\$ 7.879,60 por m^2 , seguido pelo bairro Altiplano Cabo Branco, com um valor médio de R\$ 7.218,70 por m^2 .

A análise de alguns bairros se torna limitada devido à baixa quantidade de imóveis disponíveis no momento da raspagem de dados. Por exemplo, o bairro de Barra de Gramame, que apresenta o menor valor médio de m^2 , tinha apenas um imóvel à venda no momento da coleta de dados. Da mesma forma, o bairro Jardim Veneza, o segundo com o menor valor médio por m^2 , também possuía poucos registros. Isso indica que os valores apresentados para esses bairros podem não refletir com precisão o mercado imobiliário local e podem ser superiores em outras circunstâncias.

(FALAR DAS VARIÁVEIS BINÁRIAS)

A distribuição das variáveis foram analisadas em termos do tipo do imóvel a partir de um gráfico de violino. Pela Figura 5.3, é fácil perceber que a maioria das distribuições possuem assimetria negativa. Os apartamentos possuem caudas longas à direita, indicando presença de valores extremamente altos e que indicam que talvez seja necessário a aplicação de alguma transformação para a estabilização da variância.

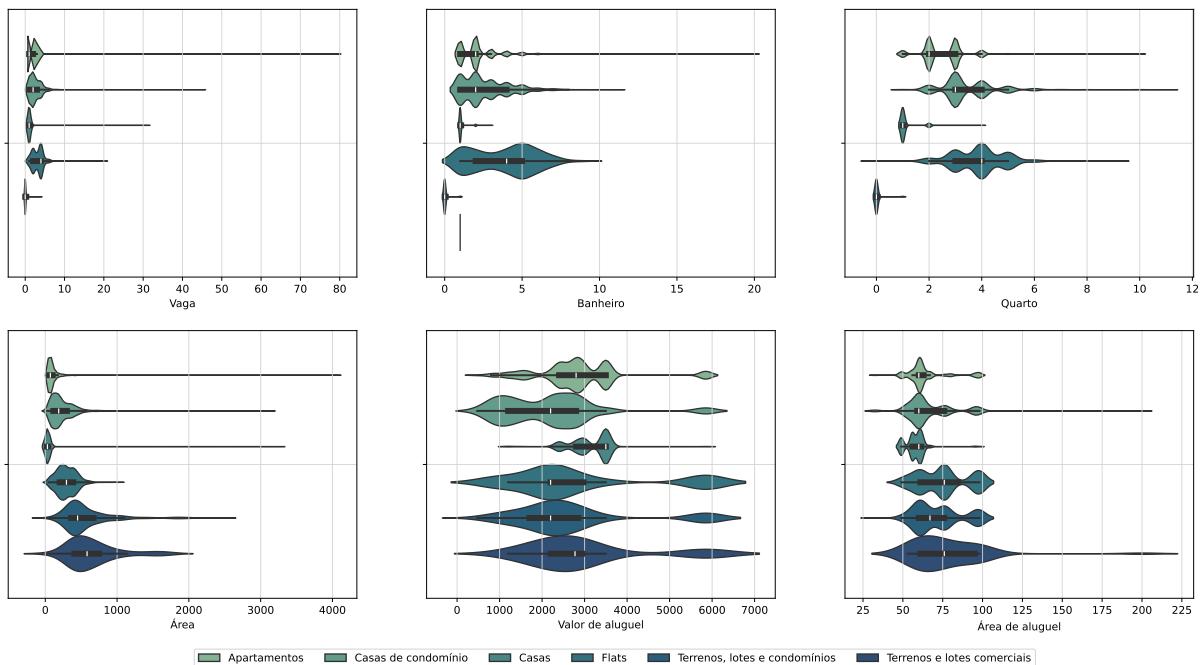


Figura 5.3: Distribuição das variáveis numéricas.

Para reduzir a assimetria da distribuição dos valores dos imóveis, foi aplicada uma transformação logarítmica. O gráfico de densidade à esquerda na Figura 5.4 mostra os dados originais da distribuição do valor dos imóveis. Há uma tendência dos valores ficarem mais concentrados em uma faixa mais baixa, mas alguns imóveis apresentam valores excepcionalmente altos, o que acaba gerando uma distribuição assimétrica positiva. Com a aplicação da transformação logarítmica, a assimetria é suavizada, comprimindo os valores mais altos. Isso tende a aproximar a distribuição de uma forma mais simétrica, facilitando a modelagem e análise estatística.

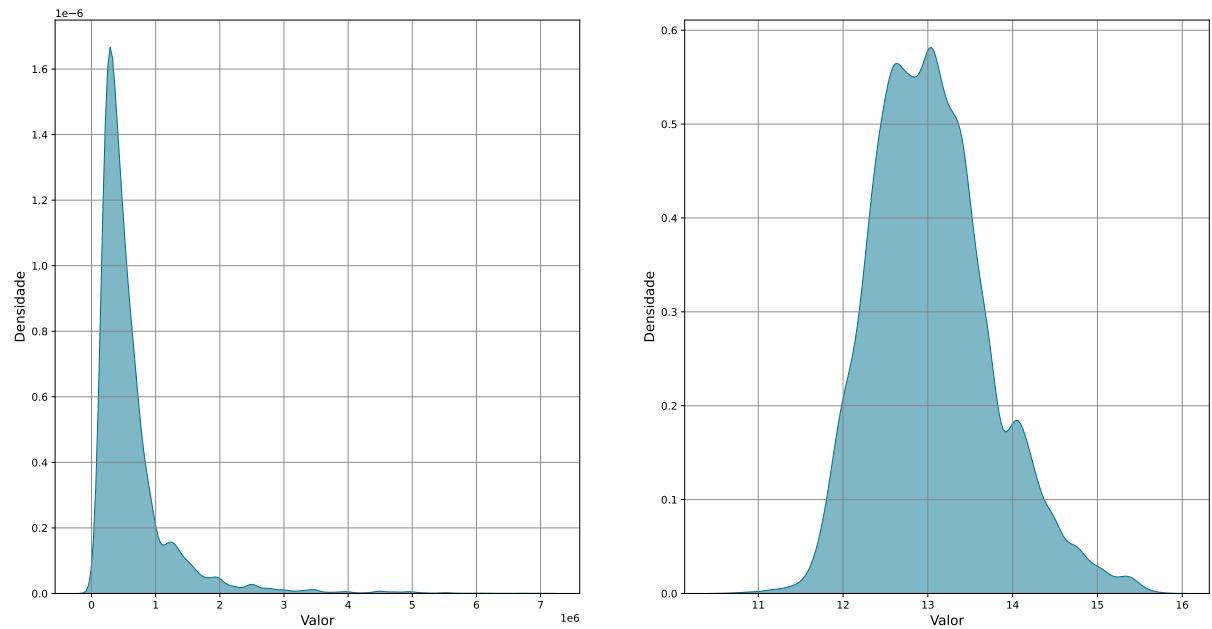


Figura 5.4: Comparaço entre distribuiao dos valores dos imveis antes e depois da transformaao logaritmica.

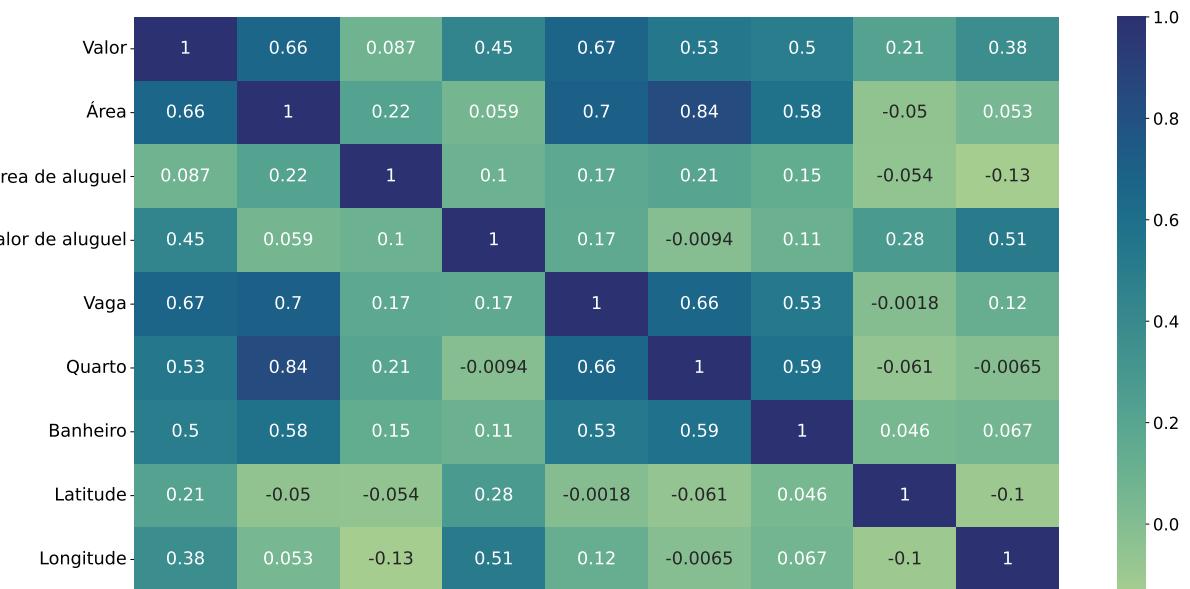


Figura 5.5: Grfico de correlaao de Spearman das variveis independentes.

A Figura 5.5 apresenta a matriz de correlação entre as variáveis numéricas do conjunto de dados. As cores mais escuras indicam uma correlação mais forte entre as variáveis, enquanto as cores mais claras indicam o contrário. O valor do imóvel apresenta maior correlação com as variáveis de área do imóvel e número de vagas de estacionamento. Além disso, o valor do imóvel tem alta correlação com o valor médio do aluguel, número de quartos e banheiros, além de ser fortemente influenciado pela localização das propriedades. Algumas variáveis apresentam multicolinearidade entre si, mas os algoritmos utilizados selecionam aleatoriamente as variáveis para a modelagem, o que reduz o risco de selecionar variáveis redundantes.

5.2 Tunagem dos modelos

Como os algoritmos utilizados neste trabalho são baseados em árvores de decisão ou podem utilizar algoritmos baseados em árvore, como Random Forest, Gradient Boosting e suas variações, os parâmetros escolhidos para otimização serão bastante parecidos. Assim, foram ajustados hiperparâmetros como o número de árvores e a profundidade das árvores, de forma a capturar a complexidade das relações presentes nos dados. Além disso, para os algoritmos baseados em Gradient Boosting, o hiperparâmetro de taxa de aprendizado também foi ajustado.

Todos os modelos e bibliotecas usados neste trabalho seguem a API do scikit-learn. Nessa API, o hiperparâmetro que define a quantidade de árvores é denominado `n_estimators`, o de profundidade das árvores é `max_depth`, e o de taxa de aprendizado é `learning_rate`. Com a configuração da função objetivo na biblioteca Optuna, foi possível encontrar o ponto ótimo desses hiperparâmetros para cada modelo. As métricas obtidas para cada algoritmo pode ser visualizado na Tabela 5.1.

Tabela 5.1: Métricas obtidas de cada algoritmo

Algoritmo	RMSE	R^2	MAPE
Random Forest	0,28972	86,78792%	0,01373
Gradient Boosting	0,28730	86,98259%	0,01377
Light Gradient Boosting	0,28493	87,17132%	0,01346
Extreme Gradient Boosting	0,28659	87,03891%	0,01340
Stacking	0,28473	87,18793%	0,01357

5.2.1 Tunagem da Random Forest

Para o modelo Random Forest, a função objetivo foi definido apenas para otimizar os hiperparâmetros de quantidade de árvores e de profundidade da árvore. O espaço de procura para a quantidade de árvores foi definido entre 1 a 1000 árvores. Já o hiperparâmetro de profundidade da árvore foi definido entre 20 a 1000. Além disso, foi utilizado aleatoriamente $m = \sqrt{p}$ das p variáveis independentes como candidatas para a divisão.

A figura Figura 5.6a mostra a variação da estatística de erro em função dos valores de cada hiperparâmetro ao longo dos trials. O gráfico Figura 5.6b exibe a importância de cada hiperparâmetro, calculada pelo método FANOVA. Na figura Figura 5.6c, observa-se

a variação do erro para cada trial, com a linha vermelha representando o menor valor obtido em cada etapa. Por fim, a figura Figura 5.6d é um gráfico de área que mostra a interação entre os hiperparâmetros tunados em relação à estatística de erro.

A partir de Figura 5.6a, é possível ver que o erro tende a ser menor para valores menores do hiperparâmetro de profundidade das árvores (`max_depth`). Em contraste, para o hiperparâmetro de número de árvores (`n_estimators`), o erro apresenta uma tendência de redução e estabilização à medida que o número de árvores aumenta. Esse comportamento também é observado em Figura 5.6d, onde valores menores para `max_depth` e maiores para `n_estimators` resultam em um modelo com menor erro de generalização.

Além disso, o gráfico de importância dos hiperparâmetros em Figura 5.6b revela que o hiperparâmetro `n_estimators` possui uma maior contribuição na performance do modelo. A figura Figura 5.6c ilustra o comportamento da métrica de erro para cada trial, mostrando que o método de otimização alcança um de seus melhores valores pouco antes do trial 20. Após esse ponto, o otimizador não consegue encontrar valores de erro significativamente menores.

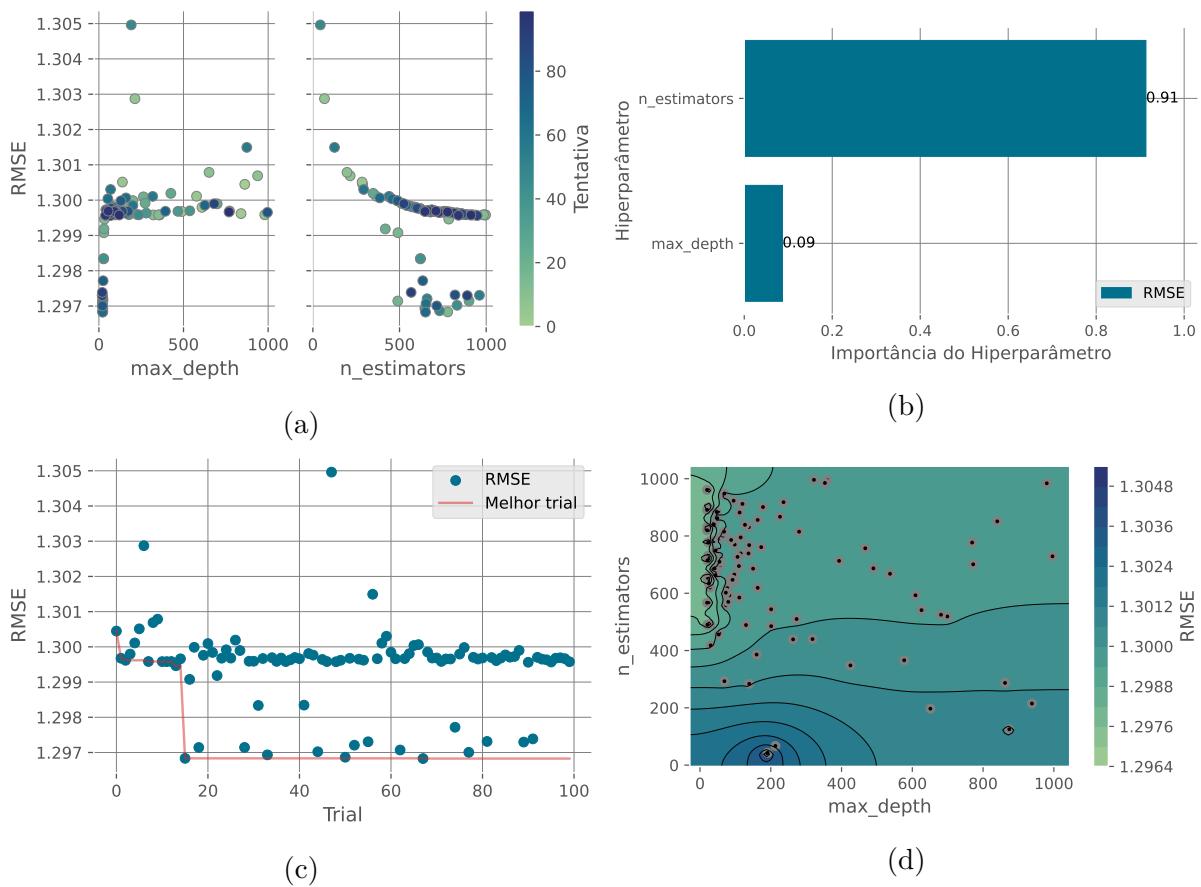


Figura 5.6: Resultados da tunagem da Random Forest.

Tabela 5.2: Melhores hiperparâmetros para Random Forest

Tentativa	RMSE	n_estimators	max_depth
67	1,29682	650	22

5.2.2 Tunagem do Gradient Boosting

Para o algoritmo de Gradient Boosting, os hiperparâmetros ajustados foram a taxa de aprendizado, a quantidade de árvores e a profundidade das árvores. No Optuna, o espaço de busca definido na função objetivo para a taxa de aprendizado variou de $1 \cdot 10^{-5}$ a $1 \cdot 10^{-1}$; para a profundidade das árvores, de 3 a 500; e para a quantidade de árvores, de 50 a 1500. Assim como no Random Forest, foi utilizada $m = \sqrt{p}$ das p variáveis independentes para realizar as divisões nas árvores.

A análise da importância dos hiperparâmetros, apresentada na Figura 5.7b, indica que o hiperparâmetro com maior influência na variação da função objetivo — e, consequentemente, na performance do modelo — é a taxa de aprendizado. Em seguida, a profundidade das árvores é o segundo mais relevante, enquanto o número de árvores tem a menor influência.

Diferentemente do modelo de Random Forest, o algoritmo de Gradient Boosting não apresentou melhorias significativas, como ilustrado na Figura 5.7c, onde a estatística de erro poucas vezes ficou abaixo de 1,3. A relação entre os hiperparâmetros é bastante similar à obtida para o Random Forest. Na Figura 5.7d, observa-se que o método de otimização TPE tende a selecionar valores menores para a profundidade das árvores e maiores para o número de árvores. No entanto, também há uma preferência por uma menor quantidade de árvores quando a taxa de aprendizado diminui.

Tabela 5.3: Melhores hiperparâmetros para Gradient Boosting

Tentativa	RMSE	n_estimators	max_depth	learning_rate
50	1,28891	1500	6	0,08730

5.2.3 Tunagem do LGBM

Para a otimização do LGBM, foram considerados os mesmos hiperparâmetros do algoritmo de Gradient Boosting, mas agora também com a tunagem do número de folhas das árvores. O espaço de busca para a quantidade de árvores foi definido entre 100 e 2000, para a taxa de aprendizado no mesmo intervalo usado no Gradient Boosting, para a profundidade das árvores e número de folhas o intervalo foi definido da mesma forma, entre 100 e 500.

As trials de otimização do modelo LGBM (Figura 5.8c) apresentaram um comportamento semelhante ao do Gradient Boosting, porém com maior estabilidade em cada tentativa e uma convergência mais rápida. Ao analisar a contribuição de cada hiperparâmetro para o desempenho do modelo na Figura 5.8b, observa-se que a quantidade de árvores é o hiperparâmetro de maior influência, seguido pela taxa de aprendizado, número de folhas e, por último, a profundidade da árvore.

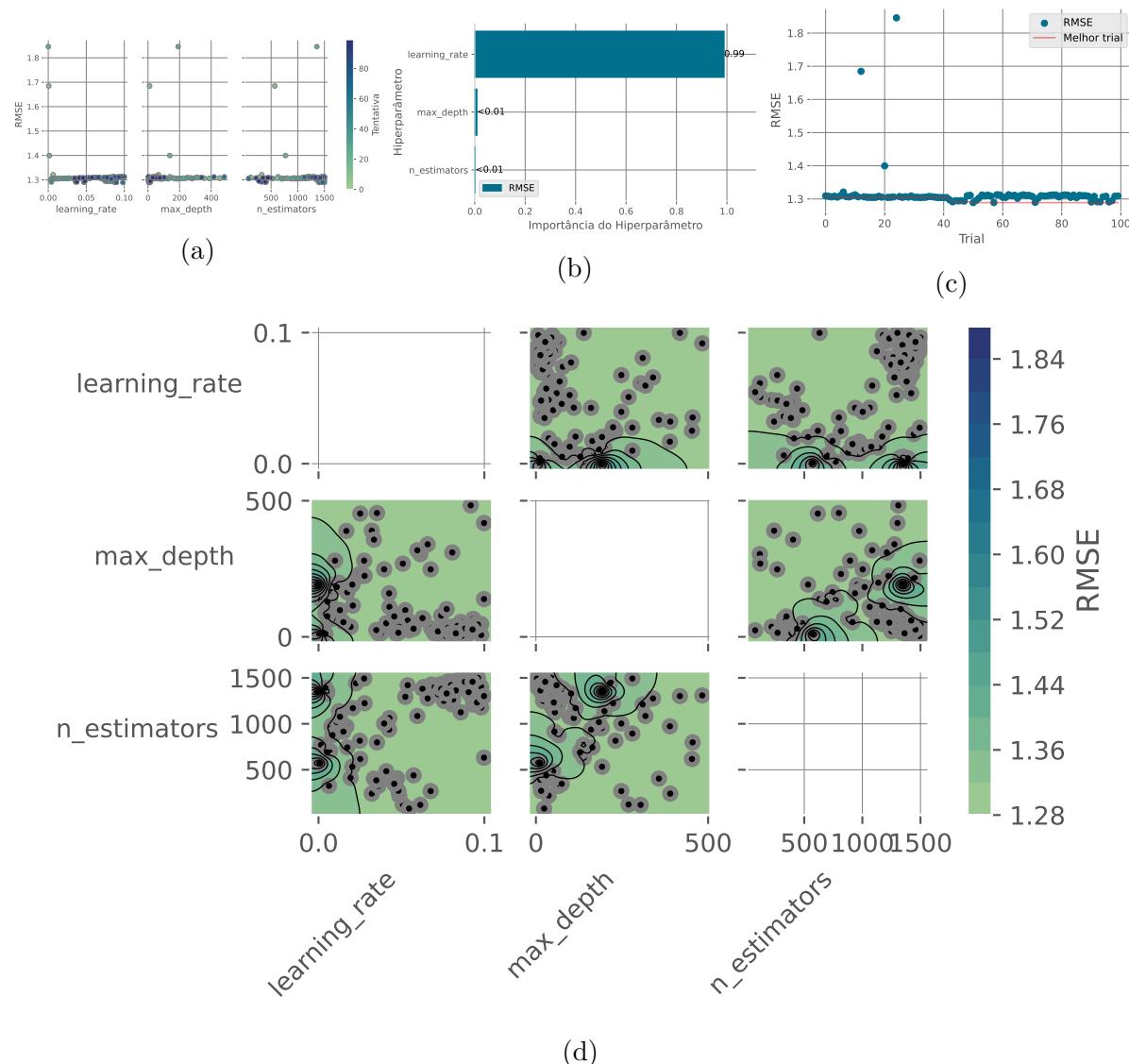


Figura 5.7: Resultados da tunagem do Gradient Boosting

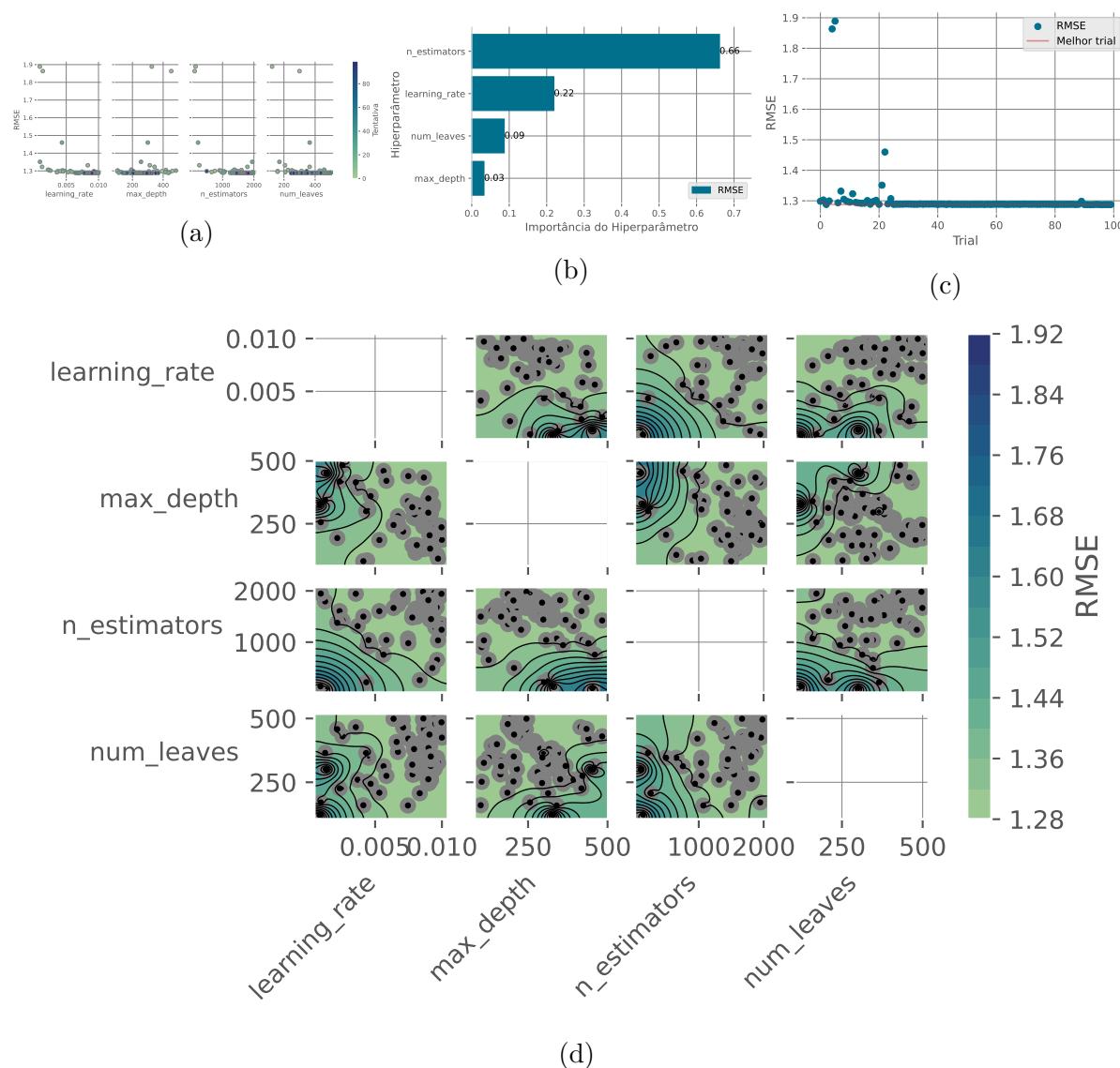


Figura 5.8: Resultados da tunagem do LGBM

Analizando a relação entre o hiperparâmetro de profundidade e o número de folhas em função da estatística de erro na Figura 5.8d, observa-se que uma maior quantidade de folhas e uma menor profundidade das árvores resultam em um modelo com erro menor. Esse mesmo comportamento é observado para os outros hiperparâmetros, uma quantidade maior de folhas combinada com uma taxa de aprendizado crescente também produz um modelo com menor erro, assim como um número maior de árvores.

Tabela 5.4: Melhores hiperparâmetros para Light Gradient Boosting

Tentativa	RMSE	n_estimators	num_leaves	max_depth	learning_rate
96	1,28722	1798	247	299	0,00903

5.2.4 Tunagem do XGBoost

Para o algoritmo Extreme Gradient Boosting, os hiperparâmetros selecionados para a tunagem foi a taxa de aprendizagem, profundidade da árvore e quantidade máxima de árvores. Na configuração da função objetivo para a tunagem dos hiperparâmetros pelo optuna, a taxa de aprendizagem variou entre $1 \cdot 10^{-7}$ e 0.5, a profundidade da árvore variou entre 3 e 50 e a quantidade de árvores variou entre 50 e 1000.

A partir da Figura 5.9b, o hiperparâmetro que representou a maior variação na função objetivo, e portanto a maior importância para a performance do modelo, foi a taxa de aprendizagem. A profundidade da árvore representou 22% da variação e por último vem a quantidade de árvores. Observando a Figura 5.9c, é possível ver que o algoritmo de XGBoost foi aquele que apresentou a menor variação entre os trials, com um único trial com erro acima de 2,0.

O algoritmo de otimização TPE apresentou uma maior frequência de procura do hiperparâmetro de taxa de aprendizagem para valores menores, como pode ser visto na Figura 5.9a. O mesmo acontece para o hiperparâmetro de profundidade de árvores, há uma repetição maior para valores menores, indicando que a estatística de erro tendeu a ter valores menores para essa região. Para a quantidade de árvores esse padrão foi diferente, uma quantidade maior de árvores faz com que o modelo tenha um melhor ajuste. Essas relações também podem ser observadas a partir da Figura 5.9d, menores taxas de aprendizado, combinado com uma menor profundidade de árvore e maior quantidade de árvores apresentam menor valor na função objetivo.

Tabela 5.5: Melhores hiperparâmetros para Extreme Gradient Boosting

Tentativa	RMSE	n_estimators	max_depth	learning_rate
81	1,28752	788	8	0,07119

A tabela Tabela 5.5 apresenta a tentativa em que a combinação de hiperparâmetros resultou no menor erro para o algoritmo de Extreme Gradient Boosting. Nesse caso, a melhor tentativa foi a de número 81, com uma configuração de 788 árvores, profundidade máxima de 8 e taxa de aprendizado de 0,07119.

Para o LGBM, a melhor combinação foi encontrada na tentativa 96, com 1.798

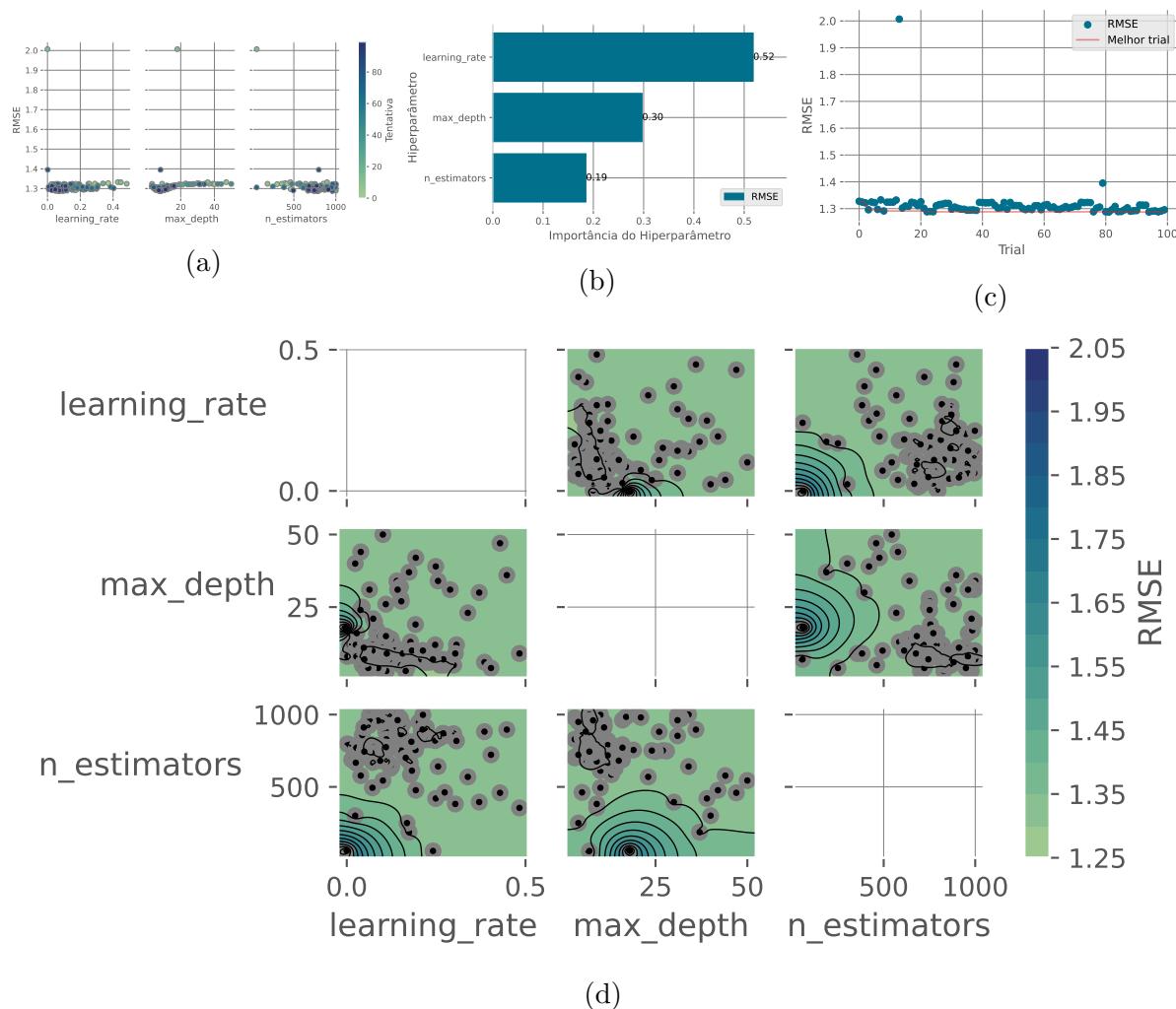


Figura 5.9: Resultados da tunagem do XGBoost

árvores, 247 folhas, profundidade máxima de 299 e taxa de aprendizado de 0,00903, como apresentado na tabela Tabela 5.4.

Por fim, para os algoritmos Gradient Boosting e Random Forest, as melhores tentativas foram as de números 50 e 67, respectivamente. No caso do Gradient Boosting, a configuração ideal foi composta por 1.500 árvores, profundidade máxima de 6 e taxa de aprendizado de 0,08730. Já para o Random Forest, a melhor combinação consistiu em 650 árvores e profundidade máxima de 22. Os resultados estão na Tabela 5.3 e Tabela 5.2, respectivamente.

5.3 Resultados dos modelos

Nesta seção, serão apresentados os resultados do ajuste de cada algoritmo utilizado: Random Forest, Gradient Boosting, LightGBM e Extreme Gradient Boosting. A análise do ajuste foi realizada com base no gráfico que compara os valores estimados aos valores observados dos imóveis. Além disso, foram utilizadas métricas de erro, como MAPE, R^2 e RMSE, para avaliar o desempenho dos modelos.

O ajuste do algoritmo Random Forest é apresentado na figura Figura 5.10a, que exibe a comparação entre os valores previstos e observados. Em termos da transformação logarítmica realizada, a raiz do erro quadrático médio (RMSE) foi de 0,28972, enquanto o coeficiente de determinação R^2 alcançou 86,79, indicando que o modelo explica 86,79 da variação dos dados. Em relação ao MAPE, o valor foi de 0,01373, assim, em média, as previsões realizadas pelo modelo Random Forest, considerando a transformação logarítmica, estiveram 1,373% distantes dos valores reais.

Em relação ao algoritmo de Gradient Boosting, o seu ajuste consegue explicar 86,98% dos dados, bastante semelhante à Random Forest, com uma diferença não tão significativa. Por outro lado, o RMSE do Gradient Boosting foi menor que a Random Forest, tendo sido de 0,28730. Observando o MAPE, não houve também muitas diferenças, embora o MAPE do Gradient Boosting tenha sido maior. O MAPE indica que as previsões geradas pelo Gradient Boosting estão 1,377% distantes de seus valores observados. Seu ajuste pode ser observado na Figura 5.10b.

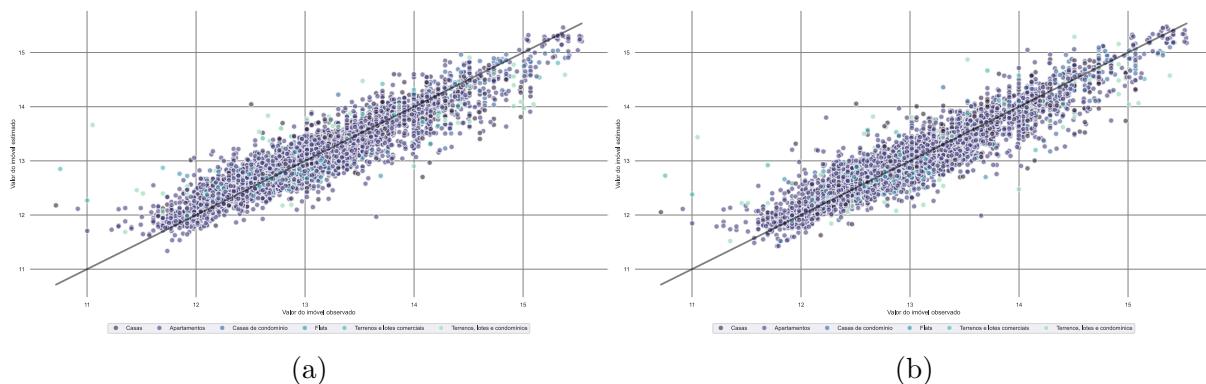


Figura 5.10: Valores previstos em função dos observados do algoritmo Random Forest e Gradient Boosting, respectivamente.

O Light Gradient Boosting conseguiu melhorias em relação aos resultados dos últimos

dois algoritmos. Em relação ao seu R^2 , o modelo consegue explicar 87,17% dos dados. O seu RMSE ficou em 0,28493 e o MAPE em 0,01346. Dessa forma, as previsões geradas pelo Light Gradient Boosting estão em média 1,346% distantes de seus valores reais. A melhora do ajuste do Light Gradient Boosting fica bastante perceptível observando a figura de seus valores estimados em função dos observados (Figura 5.11a). As previsões geradas pelos Gradient Boosting e Random Forest desviam bastante, principalmente em relação ao final da distribuição.

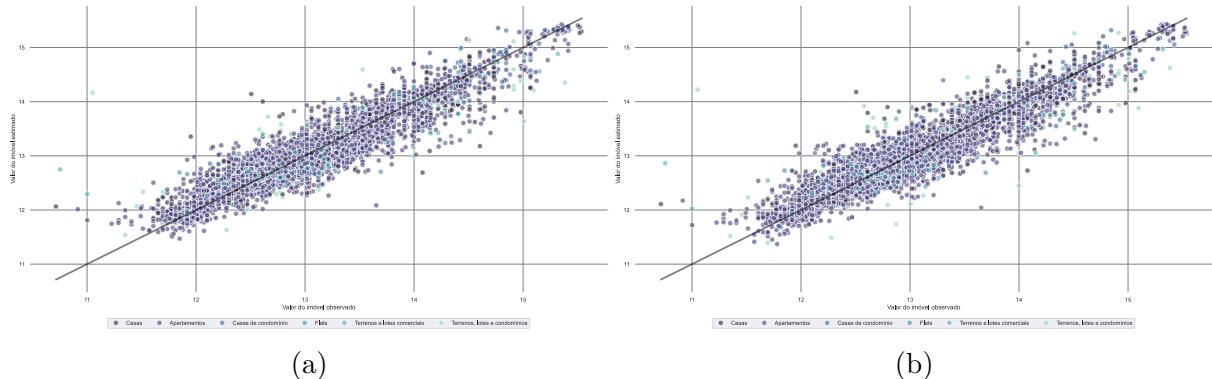


Figura 5.11: Valores previstos em função dos observados do algoritmo Light Gradient Boosting e Extreme Gradient Boosting, respectivamente.

Assim como o LGBM, o Extreme Gradient Boosting também obteve resultados melhores que os algoritmos de Random Forest e Gradient Boosting. Entretanto, obteve piores em relação às estatísticas de erro de RMSE e R^2 . O RMSE obtido pelo XGBoost foi de 0,28659 e o seu R^2 foi de 87,03891%, indicando que o modelo consegue explicar 87,03891% dos dados. No entanto, obteve um MAPE pouco menor que o LGBM. As previsões geradas pelo Extreme Gradient Boosting estão em média 1,357 distantes de seus valores reais.

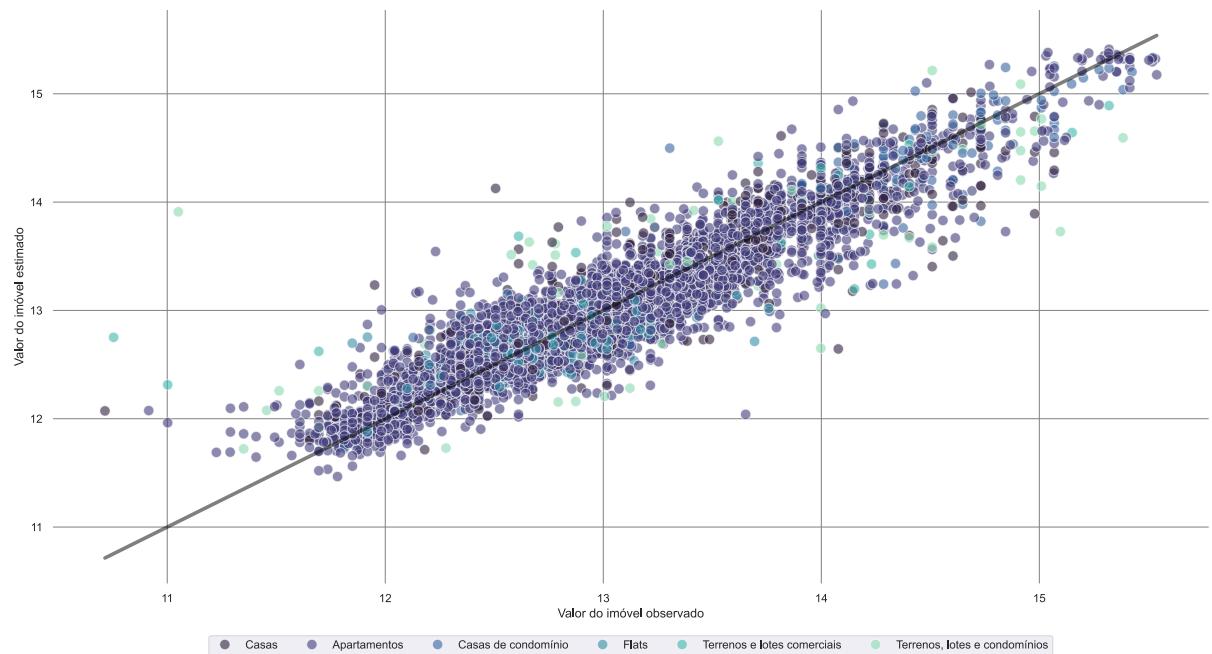


Figura 5.12: Valores previstos em função dos observados do algoritmo Stacking

Utilizando os últimos modelos com seus hiperparâmetros otimizados e utilizando o Light Gradient Boosting como estimador final, as predições foram feitas com o Stacking. As predições geradas pelo Stacking, analisando as métricas de erro utilizadas, foram melhores até mesmo que o Light Gradient Boosting. O RMSE obtido para o Stacking foi de 0,28473 e o modelo consegue explicar 87,18793%. A única métrica que não foi melhor que do algoritmo LGBM foi o MAPE, que foi de 1,357%, mas a diferença é pouca.

5.4 Impacto e importância das variáveis na predição

O impacto e a importância das variáveis na predição de modelos são fundamentais para entender quais fatores exercem maior influência sobre as previsões, neste caso, para as predições de valores de imóveis. Esta seção será dedicada à discussão do impacto das predições individuais e da importância das variáveis utilizadas na construção do modelo. A análise será feita com o modelo que apresentou os melhores resultados, o algoritmo de Stacking.

Para analisar a influência das variáveis na predição dos valores de imóveis, foi utilizado o gráfico de ICE em conjunto com a linha de PDP. Esses gráficos estão representados na Figura 5.13.

As previsões mostram um aumento claro com o tamanho da área, com o PDP apresentando uma tendência crescente consistente, alinhada à maioria das curvas ICE. Para a quantidade de banheiro, a relação é majoritariamente plana, com apenas pequenas variações. Um comportamento semelhante é observado para a quantidade de quartos, indicando uma influência limitada dessas variáveis no modelo. No caso do valor médio de aluguel, o PDP revela uma tendência ascendente bem definida, e as curvas ICE reforçam esse padrão, sugerindo que valores de aluguel mais altos estão fortemente associados a preços de imóveis mais elevados. Já a área média de aluguel apresenta um padrão mais uniforme, indicando uma contribuição menos significativa para explicar os preços dos imóveis. As curvas ICE para as coordenadas geográficas exibem um padrão mais complexo nas previsões. No entanto, é possível identificar um aumento nos valores previstos dos imóveis à medida que as coordenadas aumentam. Esse comportamento sugere que os imóveis localizados mais próximos da região litorânea de João Pessoa tendem a apresentar valores mais elevados. Por fim, a quantidade de vagas de garagem demonstra uma relação ascendente clara nas predições, evidenciando uma influência positiva dessa variável no valor dos imóveis.

A análise da importância das variáveis foi realizada utilizando o método SHAP. Diferentemente da importância de variáveis gerada por algoritmos baseados em árvores, que é calculada com base na redução da impureza ou no ganho de informação ao longo das divisões no modelo, o SHAP é baseado na magnitude dos valores de Shapley atribuídos a cada variável. Dessa forma, como pode ser observado na Figura 5.14b, a variável com maior impacto na predição do modelo é a área, a qual altera, em média, mais de 25 pontos percentuais na predição absoluta dos imóveis. A quantidade de vagas de garagem, o valor médio do aluguel e a localização geográfica também se destacam entre as variáveis mais relevantes na predição do valor dos imóveis.

Com a Figura 5.14a, observa-se um resumo da análise dos valores Shapley. A variável área se destaca como a de maior impacto no modelo, especialmente para valores elevados,

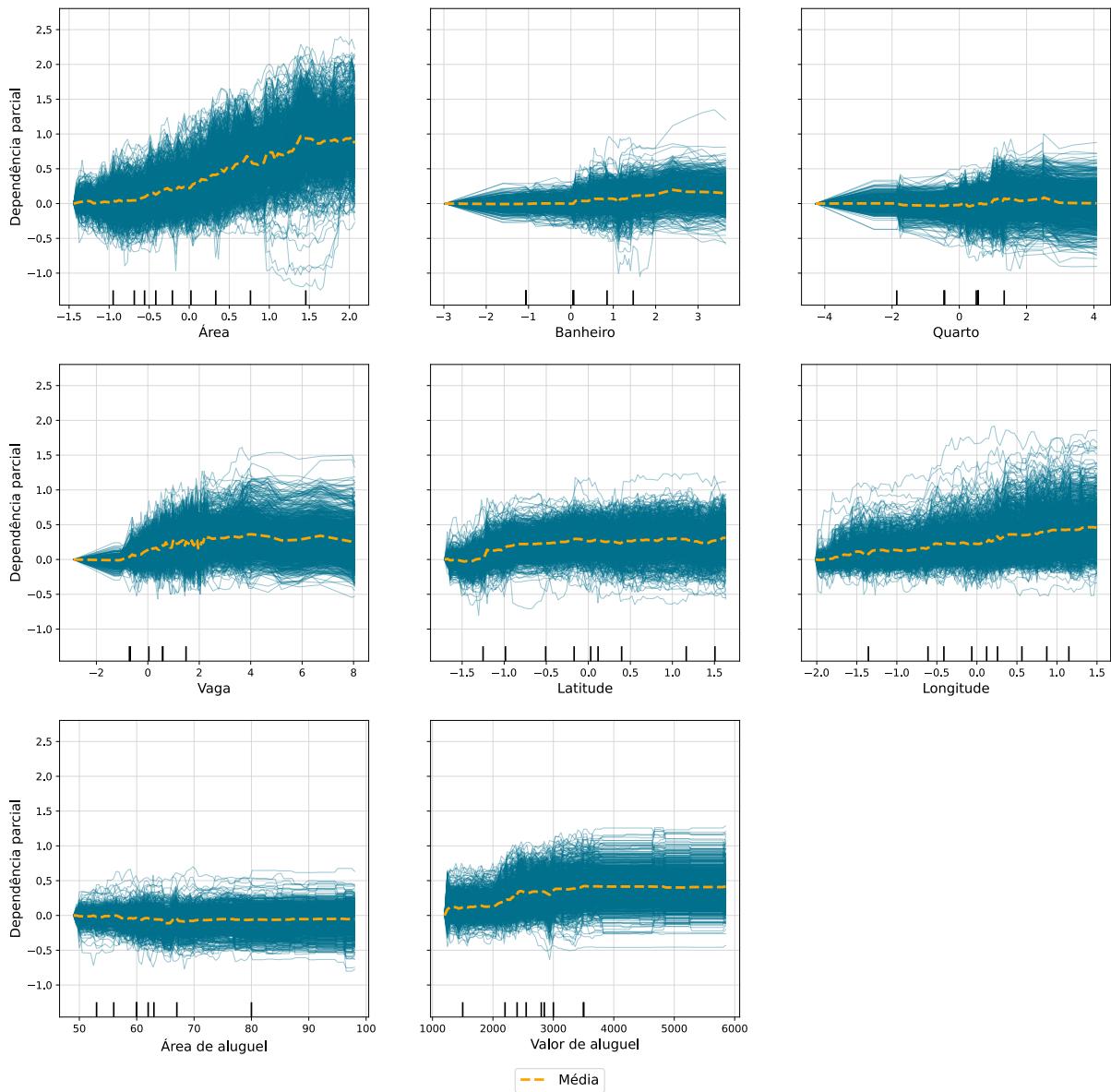


Figura 5.13: Gráfico de ICE e PDP.

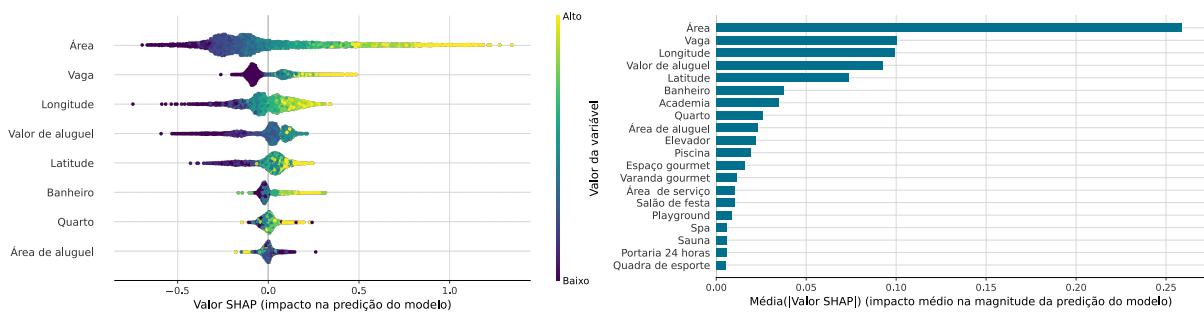


Figura 5.14: Impacto e importância das variáveis na predição a partir do método SHAP.

que aumentam significativamente as previsões. As coordenadas geográficas indicam que valores mais altos tendem a elevar a previsão do valor dos imóveis, refletindo a valorização de imóveis localizados mais próximos às regiões litorâneas de João Pessoa. Por outro lado, a variável de área média de aluguel apresenta um padrão menos linear, com valores baixos ainda gerando impacto positivo nas previsões, evidenciando uma relação mais complexa com o valor dos imóveis.

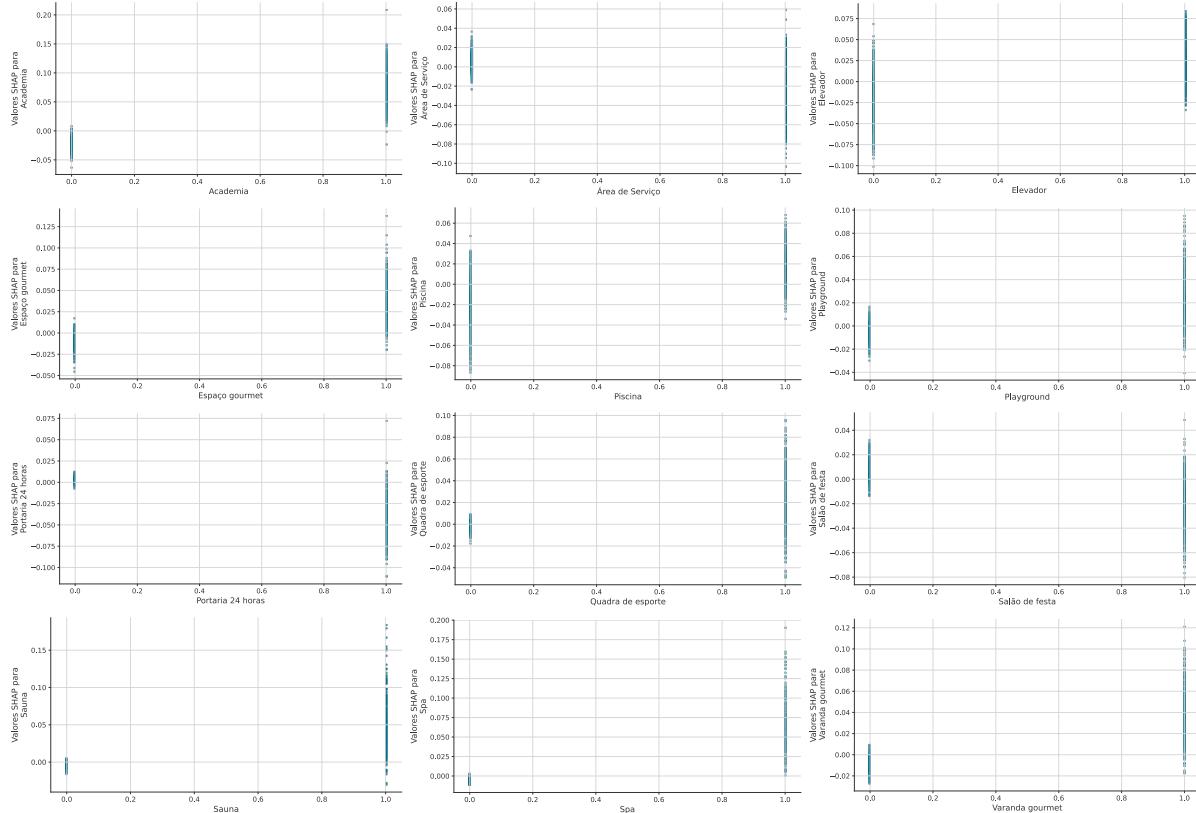


Figura 5.15: Gráfico de dependência de valores SHAP para variáveis binárias.

Para analisar as variáveis binárias, foram utilizados gráficos de dependência para cada uma delas, os quais podem ser visualizados na Figura 5.15. Ao analisar os valores SHAP para os imóveis com e sem academia, observa-se que essa variável tende a aumentar a previsão para imóveis que possuem academia. O mesmo padrão é identificado para imóveis com elevador, embora aqueles sem elevador também apresentem uma tendência de aumento na previsão, o que pode ser explicado pela influência de outras características dos imóveis. Como já mencionado na análise da importância das variáveis, sauna e spa são características com baixa importância. No entanto, ao observar seus gráficos de dependência, é possível perceber que essas variáveis podem ser úteis na avaliação de imóveis que possuem essa característica e tem valores mais elevados. O mesmo comportamento é observado para varanda gourmet, playground e espaço gourmet.

6 Conclusão

Referências

- ABECIP. **O Crédito Imobiliário no Brasil: Caracterização e Desafios.** [S.l.]: Associação Brasileira das Entidades de Crédito Imobiliário e Poupança, 2007b.
- _____. **A Revolução do Crédito Imobiliário: 44 Anos (1967–2011).** [S.l.]: Associação Brasileira das Entidades de Crédito Imobiliário e Poupança, 2007a.
- _____. **IV Prêmio ABECIP de Monografia em Crédito Imobiliário e Poupança.** [S.l.]: Associação Brasileira das Entidades de Crédito Imobiliário e Poupança, 2015.
- AKIBA, T. *et al.* Optuna: A Next-generation Hyperparameter Optimization Framework. [S.l.]: [s.n.], 2019.
- ALLAIRE, J.; DERVIEUX, C. [quarto: R Interface to 'Quarto' Markdown Publishing System](#). [S.l.]: [s.n.], 2024.
- ASSUMPÇÃO FILHO, C. A. B. O crédito imobiliário no Brasil. 2011.
- BAYER, M. [SQLAlchemy](#). Em: BROWN, A.; WILSON, G. (Org.). **The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks.** [S.l.]: aosabook.org, 2012.
- BERGSTRA, J. *et al.* Algorithms for hyper-parameter optimization. **Advances in neural information processing systems**, 2011. v. 24.
- _____; YAMINS, D.; COX, D. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. [S.l.]: PMLR, 2013. p. 115–123.
- BISCHL, B. *et al.* Hyperparameter optimization: Foundations, algorithms, best practices, and open challenges. **Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery**, 2023. v. 13, n. 2, p. e1484.
- BREIMAN, L. Bagging predictors. **Machine learning**, 1996. v. 24, p. 123–140.
- CAMBON, J. *et al.* tidygeocoder: An R package for geocoding. **Journal of Open Source Software**, 2021. v. 6, n. 65, p. 3544. Disponível em: <<https://doi.org/10.21105/joss.03544>>.
- CAMPOS, S. F. Precificação de imóveis e seus elementos agregadores de valor sob a visão do consumidor: uma análise do mercado imobiliário de João Pessoa-PB. 2014.

- CHEN, T.; GUESTRIN, C. Xgboost: A scalable tree boosting system. [S.l.]: [s.n.], 2016. p. 785–794.
- COSTA FARIAS, B. M. Da. A evolução do mercado imobiliário brasileiro e o conceito de Home Equity. 2010.
- FIPE – FUNDAÇÃO INSTITUTO DE PESQUISAS ECONÔMICAS. **Índice FipeZap de Preços de Imóveis Anunciados: Notas Metodológicas**. São Paulo: Fipe, 2024.
- FRIEDMAN, J. H. Stochastic gradient boosting. **Computational statistics & data analysis**, 2002. v. 38, n. 4, p. 367–378.
- GARNETT, R. **Bayesian optimization**. [S.l.]: Cambridge University Press, 2023.
- GOLDSTEIN, A. *et al.* Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. **Journal of Computational and Graphical Statistics**, 2015. v. 24, n. 1, p. 44–65.
- HARRIS, C. R. *et al.* Array programming with NumPy. **Nature**, set. 2020. v. 585, n. 7825, p. 357–362. Disponível em: <<https://doi.org/10.1038/s41586-020-2649-2>>.
- HASTIE, T. *et al.* **The elements of statistical learning: data mining, inference, and prediction**. [S.l.]: Springer, 2009. V. 2.
- HOSSAIN, Shammamah. **Visualization of Bioinformatics Data with Dash Bio**. (Chris Calloway et al., Org.). [S.l.]: [s.n.], 2019. p. 126–133.
- HUNTER, J. D. **Matplotlib: A 2D graphics environment**. **Computing in Science & Engineering**, 2007. v. 9, n. 3, p. 90–95.
- INC., P. T. Collaborative data science. 2015. Disponível em: <<https://plot.ly>>.
- JAMES, G. *et al.* **An introduction to statistical learning**. [S.l.]: Springer, 2013. V. 112.
- KE, G. *et al.* Lightgbm: A highly efficient gradient boosting decision tree. **Advances in neural information processing systems**, 2017. v. 30.
- KOUZIS-LOUKAS, D. **Learning Scrapy**. [S.l.]: Packt Publishing Ltd, 2016.
- LUNDBERG, S. M.; LEE, S.-I. **A Unified Approach to Interpreting Model Predictions**. Em: GUYON, I. *et al.* (Org.). **Advances in Neural Information Processing Systems 30**. [S.l.]: Curran Associates, Inc., 2017, p. 4765–4774.
- MOLNAR, C. **Interpretable machine learning**. [S.l.]: Lulu. com, 2020.
- PEDREGOSA, F. *et al.* Scikit-learn: Machine Learning in Python. **Journal of Machine Learning Research**, 2011. v. 12, p. 2825–2830.

- PYTHON-VISUALIZATION. **Folium**. Disponível em: <<https://github.com/python-visualization/folium/>>.
- R CORE TEAM. **R: A Language and Environment for Statistical Computing**. Vienna, Austria: R Foundation for Statistical Computing, 2024.
- RIBEIRO, M. T.; SINGH, S.; GUESTRIN, C. "Why should i trust you?" Explaining the predictions of any classifier. [S.l.]: [s.n.], 2016. p. 1135–1144.
- SHAPLEY, L. S. A value for n-person games. **Contribution to the Theory of Games**, 1953. v. 2.
- SNOEK, J.; LAROCHELLE, H.; ADAMS, R. P. Practical bayesian optimization of machine learning algorithms. **Advances in neural information processing systems**, 2012. v. 25.
- ŠTRUMBELJ, E.; KONONENKO, I. Explaining prediction models and individual predictions with feature contributions. **Knowledge and information systems**, 2014. v. 41, p. 647–665.
- TEAM, T. Pandas Development. **pandas-dev/pandas: Pandas**. Zenodo. Disponível em: <<https://doi.org/10.5281/zenodo.3509134>>.
- THORPE, A. **selenium: Low-Level Browser Automation Interface**. [S.l.]: [s.n.], 2024.
- VAN ROSSUM, G.; DRAKE JR, F. L. **Python reference manual**. [S.l.]: Centrum voor Wiskunde en Informatica Amsterdam, 1995.
- VIRTANEN, P. *et al.* **SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python**. **Nature Methods**, 2020. v. 17, p. 261–272.
- WAGNER, F. E.; WARD, J. O. Urbanization and Migration in Brazil. **The American Journal of Economics and Sociology**, 1980. v. 39, n. 3, p. 249–259.
- WASKOM, M. L. **seaborn: statistical data visualization**. **Journal of Open Source Software**, 2021. v. 6, n. 60, p. 3021. Disponível em: <<https://doi.org/10.21105/joss.03021>>.
- WICKHAM, H. **httr: Tools for Working with URLs and HTTP**. [S.l.]: [s.n.], 2023.
- _____. **rvest: Easily Harvest (Scrape) Web Pages**. [S.l.]: [s.n.], 2024.
- _____. HESTER, J.; OOMS, J. **xml2: Parse XML**. [S.l.]: [s.n.], 2023.
- YANG, L.; SHAMI, A. On hyperparameter optimization of machine learning algorithms: Theory and practice. **Neurocomputing**, 2020. v. 415, p. 295–316.