Perfect Comps

Identifying Comparable Real Estate Properties using Machine Learning

R. H. van Heukelum
G. J. W. Oolbekkink
M. Wolting

GeoPhy
Q4 2017 - 2018

Perfect Comps

Identifying Comparable Real Estate Properties using Machine Learning

by

R. H. van Heukelum
G. J. W. Oolbekkink
M. Wolting

to obtain the degree of Bachelor of Science
at the Delft University of Technology,

Project duration:	April, 2018 – July, 2018	
Thesis committee:	Dr. Ir. C. C. S. Liem,	TU Delft, supervisor
	Ir. O. W. Visser,	TU Delft
	Ir. S. M. Mulders,	GeoPhy

TUDelft

Abstract

For traditional, manual real estate appraisals, the appraiser is required to provide a number of comparable properties (the 'Comps'). These comps act as a benchmark for the valuation as well as a provider of context in the final appraisal report. Traditionally, these comps are selected manually by an appraiser based on recent transactions within a ten mile radius. This manual selection is biased by the appraiser's market knowledge and the amount of transactions in the area. To replace this process, we developed an automated comparable selection service that does so based on objective characteristics, without restricting itself to a small spatial and/or temporal slice of the market.

Comparable selection does not have an objective ground truth, which complicates or even prohibits the use of many machine learning algorithms that could otherwise have been used. Additionally, the outcome of the service needs to be explainable to its users — it cannot be completely opaque. Finally, our service needed to integrate with a streaming data platform, with incremental new data that arrives continuously and needs to be incorporated into the service's model and output.

Our research phase focused on three aspects: determining the possible algorithms for selecting comparable properties given the constraints of explainability and a streaming environment, how to explain the output of the chosen algorithm to the user, and how to build a service around the chosen model that consumes a stream of input data and can generate a set of comparable properties ad-hoc.

Our process was based on agile methodology, with two-week sprints in which we gradually expanded our service into a fully functional proof of concept. Challenges were encountered while developing the logic to incrementally construct a database of real estate properties from the stream of data. These were resolved by switching from a document store to an RDF database, which better matched the flow of data coming in.

The final product consists of several microservices, each of which handles part of the problem domain and can be scaled out independently. A REST API and a web front-end are accessible to its users. The system was tested using both unit tests and end-to-end testing, whereas the model was refined by scoring output on closeness of features indicative of similarity.

As a future improvement, the current model used by the service is fairly simple and can most likely be improved upon once more data is available. Additionally, due to the lack of a ground truth it will be important to tune both comparable selection and explanation metrics in response to user testing.

Preface

In the past quarter we have had the pleasure to work for the company of GeoPhy. It has been a great experience and an amazing learning process. In this document you will find a condensed overview of our activities for the service that we have developed: the perfect comps system. The document has been written in a chronological order, taking the reader through our thought process and showing any encountered obstacles.

We would like to thank GeoPhy, and especially Sander, for giving us this amazing opportunity. We would also like to thank Cynthia for keeping us critical and showing interest in our activities. This project has given us an insight into the business of software development. We have seen some of the challenges that need to be tackled in the development of production-ready Machine Learning algorithms and the services that use them. This experience we will carry on with us into our jobs or Master studies.

R. H. van Heukelum
G. J. W. Oolbekkink
M. Wolting
Delft, June 2018

Contents

1 Introduction

Data can hold treasures, mining it can reveal great surprises. Data enrichment and mining are at the core of GeoPhy's business. They focus on data-driven real estate appraisals. This is a process that requires some major engineering steps. Over the past years, the business has grown from being a disruptive innovation start-up to being a company that employs over 60 people in two continents that aims to change the real estate business on a much larger scale.

GeoPhy has given us the opportunity to work on a new project in the business of real estate and big data. We have been working on the so-called Perfect Comps (comparables) service. It is aimed at exploring the possibilities and applications of machine learning in identifying sales comparables for properties. Perfect Comps is meant as a gateway product for the potential clients of GeoPhy. Their Automated Valuation Model (AVM) is, in many cases, still too advanced and too big of a step for the conservative industry. Therefore, we have been tasked to find ways to make this new product possible. In order to document our work and our process, we have come up with a research question with several sub-questions:

How can we design a service that accurately identifies sales comparables?

How can we make it scalable?

How can we explain the results?

How can we integrate our service into GeoPhy's infrastructure?

In our project we worked in two main phases: research and implementation. In this document, we have combined both phases to give an overview of our work here at GeoPhy. This thesis will go over the research phase in chapter 2 and chapter 3. In those chapters, we explore the problem and evaluate the available literature on it. chapter 4 deals with the architecture that we designed for our service, and in chapter 5 we give an overview of the details of the actual implementation. This thesis also considers the challenges that we have had to overcome in realizing our product. chapter 6 discusses our process and results, with a reflection on how they fulfill the requirements. It also discusses the ethical implications of the product. We conclude our work in chapter 7. Finally, to give concrete feedback to GeoPhy, we have incorporated several recommendations in chapter 8.

2 Problem Description and Analysis

2.1. Description

The world of real estate valuations is experiencing a shift from manual appraisals to automated valuations based on data-driven approaches (Kok et al., 2017). GeoPhy is working hard to accomplish this transition within what is, generally speaking, a conservative industry. Their main product is the Automated Valuation Model (AVM) that they offer to companies to be able to generate automated valuations for their properties. This requires the people to trust the Machine Learning driven algorithms to compute the value of a certain property. This can be a step that traditional companies are scared to take.

Pagourtzi et al. (2003) state that the sales comparison approach is one of the most widely used approaches for manual valuation. It involves selecting comparables (i.e. highly similar real estate properties) and using their transactions to valuate the original property. Downie and Robson (2008) conclude that many appraisers are afraid to use AVMs due to its, generally speaking, inexplicable way of working. The structural users of AVMs say that they use them for other purposes as well (see figure 2.1). GeoPhy has set its goal to implement a new product. This will be able to give clients a view on the innovative approach that the company is providing in their main product. Figure 2.2 shows many respondents that are convinced of the fact that appraisers have an advantage because of their ability to evaluate comparables. This supports the intuition stated by GeoPhy: we have to educate the clients about the advantages that AVMs can bring.

This means that we have to convince potential customers of the power of Machine Learning in the field of real estate. Using such techniques, GeoPhy wants to empower the customers by offering a direct alternative for the traditional appraiser workflow which is instant and more accurate. This is the mission of the company [1]. While realizing this mission, they have concluded that customers need an intermediate step before diving

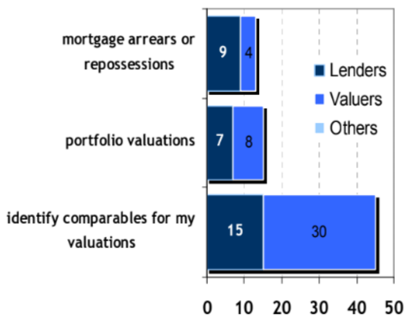

Figure 2.1: Graph taken from Downie and Robson (2008). It shows the AVM usage for other purposes among 92 respondents that have used AVMs. A significant portion of these respondents responded that they use AVMs to identify comparables.

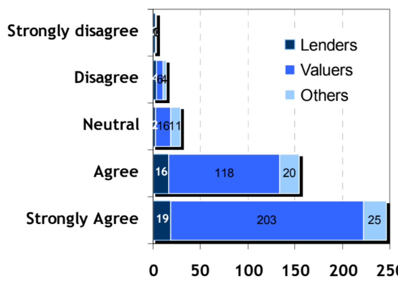

Figure 2.2: Graph taken from Downie and Robson (2008). Statement: Appraisers' ability to evaluate comparables is a major advantage over AVMs.

[1] https://geophy.com/about

into the world of fully automated valuations.

This will be offered in a new product: the perfect comparable system (perfect comps). The product will compare a subject property to the data available to the company, and generate a list of property sales comparable to the subject. Using the perfect comps product, customers can be offered a different view on the usage of machine learning. The product employs the fact that it does not necessarily need to regard the geospatial and temporal localities that traditional appraisers are bound to. For example, the market that was present 3 years ago in a different part of the country might be very similar to the market nowadays. This can, most likely, only be discovered using the data available to GeoPhy. GeoPhy has a large database of properties which is constantly updated with new data and enriched with data from all kinds of different sources.

This idea raises interesting questions: how do people find comparables? What makes a property a comparable? Training data is valuable in this stage of development. Therefore, we must look for sources to obtain it from in order to build the model. We could either look for existing comparable sets, offered by traditional appraisers for example, or utilize e.g. clustering algorithms to identify comparables ourselves. In this research report, we explore relevant fields of study to find useful techniques that we might be able to employ. This is a necessary stage in order to build a training set for our model.

2.2. Previous approaches for usage of comparables

In historical scientific literature, there are several approaches to finding comparables. The first real automation of this problem was seen in Van Dell (1991). The paper proposed a mathematical framework to select and score comparables. The selection was subsequently used in a different formula to calculate their relevance in the valuation of a certain property.

The paper by Detweiler and Radigan (1996) proposes an even more systematic and programmed approach. This is one of the first actual AVMs that we could find. It employs the strategy of finding and utilizing comparable sales to come up with a valuation. However, much of the work is still done by the appraiser that uses the tool.

Todora and Whiterell (2002) further develop the concept of an AVM. The authors explain the ideas and the choices behind their product: appraisal vision. It is one of the first commercial variants of an AVM. They explain the formulas that make up their model and the data they incorporated. Their approach is geared towards the local market they are active in. Therefore, this product can be seen as the automated variant of a local appraiser.

More recent studies (Shih-MingYou and Chang, 2009) have attempted to move away from the mathematical basis used by the aforementioned historical research. Instead, the researchers try to develop a model that has weights to compare certain properties and uses this information to construct a valuation.

Where GeoPhy tries to innovate is through the use of enriched data sets that are not bound to time and space. These data sets contain hundreds of features for each property. Therefore, there are many more potential influences that help make a comparable a accurate comparable. In the historical literature, there are no approaches that use this amount of data for the task of valuation or comparable selection. In the modern era Machine Learning algorithms can greatly enhance the amount of comparables to choose from when compared to traditional appraisal methods.

2.3. GeoPhy System Architecture

GeoPhy is working to improve the overall architecture of its central data processing system, in order to support their products' growing needs in that area. With this new architecture in place, all products will utilize a single environment that processes, link, and enriches all incoming data, and is able to scale with the company's ever-growing need for data. In this section, we will describe this architecture at a high level.

The company is meant to scale in terms of products as well as data input. GeoPhy has therefore chosen to base its products on a novel combination of RDF[2] databases (Klyne and Carroll, 2006) and Apache Kafka (Kreps et al., 2011). The architecture is centered around a continuous stream of data. Whenever new data points are added to the central RDF database, they automatically get published on a Kafka topic. We cannot consider these data points to be complete or consistent within the system, as new data is split into multiple messages and used to compute several aggregated attributes. The consumers that listen to these messages will be notified that data of a certain type has been added. This means that the added data will propagate through the system with minimal latency, which is the design motive of Apache Kafka. The data is then

[2] *Resource Description Framework*. See `https://w3.org/RDF/`

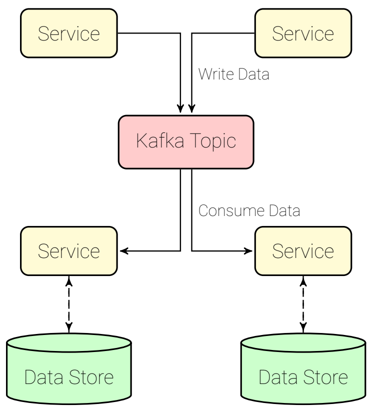

Figure 2.3: Example diagram of a Kafka architecture

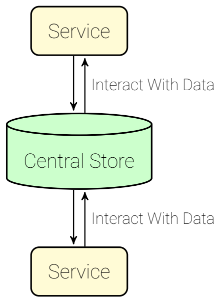

Figure 2.4: Example diagram of a centralized architecture.

processed by services and stored into different RDF databases and other database systems. Services and databases can in turn publish to other Kafka streams that other services can listen to as well. This also allows for separate data aggregation (or enrichment) that is fed back into the central RDF database.

This architecture (See Figure 2.3) provides a lot of flexibility in the addition of new services, which is different to a system where every service interacts with a central store as in Figure 2.4. It does mean that we have to consider the topics that we need in order to complete the task of finding the comparables. Another point to consider is that data is always moving. We cannot assume that the data has been fully processed, unlike systems with centralized state (e.g. a central RDBMS[3]) where tables are locked while they are being processed. This has implications for the system that we are building.

2.4. Requirements

2.4.1. Functional Requirements

Requirements that define how the product will function and what should be possible with the product.

Maximize similarity The product should generate comparables that are the most similar to the input property. Feature values in comparables can be greater or smaller than those of the input property. Similarity should be based on available comparables and feedback, and may change over time as new data flows in.

Weighted factors The product should weigh certain factors over others, in a way that is logical to its users. For example, similar market conditions but a small difference in building age should have a larger impact than different market conditions but the exact same building age.

Similar characteristics Selected comparables should have similar property characteristics, location characteristics, and market characteristics. Interest rates, economic indicators, demographics, and enriched data should be taken into account along with property information. Manually generated comps use time as a proxy for market conditions. In contrast, our model should not be constrained to recent transactions if market conditions are similar.

Explainability The result should contain an explanation for the end user. In order to improve trust and adoption rate, the user should be informed about how a specific choice was made. The distinction between property, location and market characteristics can be used to convey an explanation.

2.4.2. Non-Functional Requirements

Requirements which are imposed by the environment in which the product will be implemented.

Streaming Environment The product will run in a streaming environment, where new data can arrive often and at irregular intervals.

[3] *Relational database management system*

Incremental Data The received data arrives in parts, due to the nature of the streaming environment. There are no clear boundaries that indicate when a specific record starts or ends. New data can also update or invalidate existing data. The product should be able to process information that is still incomplete, without knowing which attributes still need to arrive.

Sparse Data The received data can be sparse. This should be taken into account when finding matches. Missing fields should not have a large negative impact on the quality of matches.

Speed The results should be generated in near-realtime. This will require preprocessing such that the results can be generated in a reasonable time.

Scalability The product needs to be set up in a way that is scalable. The infrastructure is built to allow and encourage parallel processing of messages. This means that the product should be built in such a way that it is possible to horizontally scale the system.

3
Literature Review

3.1. Comparable property selection

The primary question we need to answer to fulfill the requirements is: how do we select high-quality comparables from the available data? Comparable selection is a problem that has properties similar to both recommendation systems and structured data search. A well-defined set of properties is associated with each real estate property/transaction in the data set, in which we need to find n entries that are (approximately) the most similar to a specified property. The reason for selecting a given comparable also needs to be expressible to the service's clients to help them trust our system.

When viewed as a recommendation system, our problem suffers from a lack of training data that many existing algorithms rely on: a collection of sample items with associated recommendations. Some data in this shape exists, but is human-generated and (because of industry convention) biased towards geospatially and temporally local comparables. We explicitly want to avoid this, as a core motivation for building the product is generating better comps by evaluating a larger subsection of space and time. Generating a sufficient volume such data ourselves is infeasible, and a recommendation algorithm would therefore need to either not require training data, or be unsupervised.

When viewed as a search problem, the fields of a search query (property) cannot be assumed to exactly match existing entries, even when they are highly similar. A search algorithm would therefore need to identify entries with field values that closely resemble those of the query as well. Different fields also carry different weight, which the algorithm would need to account for.

3.1.1. k-Nearest Neighbor search (k-NN)

One versatile method of finding closest comparables is to map our feature set to a multidimensional space, and use nearest neighbor search to identify the matches closest to a query. The result quality we could achieve this way is relatively independent of the search algorithm itself, but instead defined by the quality of the spatial mapping we develop.

3.1.1.1. Preprocessing

A naive approach would be to simply transform all our features to a numerical format and then execute our search. However, since we are dealing with many features (70-700), this would suffer from the curse of dimensionality, as the search space would be sparsely populated. In addition, non-binary categorical features need to be mapped in a way that expresses the distances between them. A simple incrementing index would not suffice, for example.

To cut down on the number of dimensions, we could mark certain features as discriminative — for example, never compare homes with gardens to homes without them. Those features can then be filtered on before a k-NN search, and left out of the search space. Maximally discriminative features or constraints could also be identified using a clustering algorithm instead of doing so manually.

Finally, certain features are bound to have vastly different importance to the end result (only large differences would really matter), or be correlated with others. In our problem domain, this is supported by Diaz III (1990), where it is observed that experts tend to consider only a small subset of available information on each property when looking for comparables. In other words, there might be an opportunity for dimensionality reduction[1]. This can either be done using a linear or non-linear transformation, and many different methods exist. An important characteristic of any method we would choose is the availability of interpretable and tunable weights — we need to be able to trace back the impact of certain features to the outcome, and adapt it in response to client feedback. Examples of methods are:

Principal Component Analysis (PCA) (Pearson, 1901) and specifically weighted PCA (Yue and Tomoyasu, 2004) use a linear transformation matrix to convert the original feature space into a less correlated one. Performance would depend on how correlated the different features are, and weighted PCA allows for some tuning to steer it. As an extension to PCA, methods known as Kernel PCA (Schölkopf et al., 1998) can address non-linear feature space by employing various kernel functions to achieve linear separability.

Singular-value decomposition (SVD) (Klema and Laub, 1980) factorizes a matrix into singular values, left- and right-singular vectors. Multiplying them back together results in the original matrix. However, the original matrix can be approximated at a lower rank (i.e. less dimensions) instead by leaving out the bottom singular values at reconstruction time. Castelli et al. (2003) suggests a method based on a combination of clustering and SVD that can be used for approximate similarity search in a high-dimensional space. Performance of SVD by itself is unlikely to be worth much due to its collective nature, but CSVD is more promising due to the ability to tune the clustering algorithm.

Manifold learning is a class of non-linear methods, similar to the aforementioned ones in some of its strategies but generalized to recognize non-linear structure for improved reduction quality. Included in this category are such methods as Isometric Mapping (Tenenbaum et al., 2000), Locally Linear Embedding (Roweis and Saul, 2000), and Multidimensional Scaling (Buja et al., 2008). Non-linear methods are worth exploring if linear ones do not suffice.

Auto-encoders (Wang et al., 2016) consist of a multi-layer neural network, the layers of which first contract and then expand in width. They are trained to generate output values equal or close to the input (i.e. reconstruction). Once training has completed, the output of the narrowest layer can be used as a compressed representation of the original value. This compressed vector does not automatically preserve the spatial structure present in the data, but can be steered in such a way that it does. Options for this are contractive auto-encoders (from Rifai et al. (2011), which penalize a large output space), denoising auto-encoders (from Vincent et al. (2008), which penalize large output changes in response to small input variation), generalized auto-encoders (from Wang et al. (2014), which penalize changes in distance to other properties), k-sparse auto-encoders (from Makhzani and Frey (2013), which ignore all but the k highest activation weights in a linear encoder) and similarity-aware auto-encoders (from Chu and Cai (2017), which penalize changed output of a clustering algorithm). Auto-encoders expose many tuning options compared to previously mentioned options, and can act as generalizations of either linear or non-linear dimensionality reduction algorithms depending on chosen loss function. However, interpretability leaves to be desired, especially in deep and/or non-linearly activated auto-encoders.

Locality-preserving hashing (Indyk et al., 1997) uses a set of hash functions in such a way that they preserve distance information (close hash values imply close originals). The hash functions can also weigh features. However, collision probability can become somewhat high in higher-dimensional spaces.

It is possible to classify features into multiple silos, each of which represents an independent feature space by some semantic distinction (e.g. location, property, market). If we properly scale these feature spaces so that distances can be properly compared, we can then pick comps that minimize a weighted sum of distances to the query in each feature space. Each of the individual distances represents a feature score that we can expose, which provides an abstracted level of explainability even if any preceding dimensionality reduction method does not. See section 3.2 for more information on this.

3.1.1.2. Distance Metrics

There are several different ways to assess distance between the vectors generated by the preprocessing algorithm.

[1]Compressing high-dimensional data into a lower number of dimensions

Manhattan (taxicab) distance measures the sum of differences on every dimension between two points. It does not differentiate between e.g. a small & a large distance versus two medium distances.

Euclidean distance measures the length of a straight line between two points. It thereby takes differences on every axis into account equally, in proportion with the magnitude of it, which causes large distances to be penalized relatively more than small distances (as opposed to the Manhattan distance).

Chebyshev distance measures the maximum difference on a single dimension between two points. This makes it sensitive to outliers on a single axis. If this can be avoided during preprocessing, it might be an effective way to select comparables where every feature single is somewhat close to the queried property, as opposed to the aggregate of all features.

Mahalanobis distance measures the distance between two points as quantified by the number of standard deviations on every single axis. It requires knowledge of the covariance structure of the data set, but through that is capable of taking into account the distribution of data into measurements. In a streaming environment and with high-dimensional data, however, keeping track of covariance structure might prove expensive.

Cosine distance measures the angular distance between two points as perceived from the origin. It is less focused on the magnitude of various data points than it is on how they are oriented spatially. Magnitude would intuitively be quite meaningful in the case of comparable selection, which would most likely disqualify the cosine distance.

Gower similarity can take into account different types of features (continuous, binary, categorical, etc.) and combine them into a single similarity score. This could be interesting if our input data contains important non-discriminative binary/categorical features.

3.1.1.3. Search

Once all features have been properly encoded, they are ready for use in a k-NN algorithm. However, it would be impractical to recompute nearest neighbors for every input query separately, as it would require a number of distance queries proportional to the size of the entire data set. It therefore makes sense to apply some form of caching/indexing to the search space. Several solutions are possible, including:

k-d trees (Bentley, 1975) are constructed by recursively halving the search space along a single axis. The construction algorithm rotates through the different axes whenever it descends to a new level. The canonical implementation of a k-d tree splits along the median point on the selected axis (constructing a separating hyperplane), which will result in a balanced tree with O(log n) worst case lookup behavior. Online k-d trees are a variant of this, where element insertion/removal is possible after construction. One implementation of this is FLANN (Muja and Lowe, 2014), which naively inserts elements into the tree, and rebalances it whenever the number of elements has doubled since the last balancing. Jo et al. (2017) improved on this by reconstructing the tree based on a quality metric instead, in order to guarantee more consistent tree balance.

Ball trees (Omohundro, 1989) are similar to k-d trees, but demarcate tree leaves using hyperspheres instead of hyperplanes. Construction is more or less the same as for the k-d tree (and an online variant therefore feasible). Ball trees are supposed to perform better than a k-d tree in high-dimensional contexts (Kibriya and Frank, 2007), although performance characteristics are largely dependent on how points are spread over the search space.

(Hybrid) spill trees (Liu et al., 2005) are binary tree structures where children of a node do not have to be disjoint (i.e. left and right sub-trees can overlap). A spill margin around the chosen separation hypershape is used during construction, and any point with a distance to the dividing barrier lower than the spill margin is added to both child nodes. That way, backtracking is less likely to be required while finding nearest neighbors, which tightens worst-case performance bounds. Hybrid spill trees are spill trees with a maximum overlap threshold — if too many nodes are in the spill zone, overlap is not allowed. Tree depth can be lower as a result, depending on data set composition.

R-trees (Guttman, 1984) are a disk storage-optimized tree structure, and often employed for spatial indices in databases. They consist of recursively constructed bounding boxes that can overlap. R-trees preserve balance upon insertion. A special case of R-trees uses leaves of size k (Yang et al., 2014), so that approximate k-nearest neighbors can be found instantaneously (assuming proper construction), as leaves are allowed overlap.

R*-trees (Beckmann et al., 1990) are R-trees that use a more complicated splitting heuristic and insertion-time reconstruction to guarantee better lookup performance, at cost of more computation-intensive construction/maintenance operations.

R+-trees (Sellis et al., 1987) are R-trees that disallow overlapping internal nodes to ensure that only a single path needs to be followed at lookup time. Construction and maintenance are more complicated, but query performance is better.

M-trees (Ciaccia et al., 1997) are a (hyper)spherical variant of R-trees. They are more complicated to construct and tend to overlap more, but tend to perform better at high dimensionality much in the same way that ball trees do compared to k-d trees. M-trees also preserve balance upon insertion.

Neighborhood caching (Tang et al., 2016) involves precomputing and storing the k nearest neighbors of all inserted elements. The cached neighbors can in turn be used to help construct the initial set of neighbors for newly inserted elements. The cache could be stored as a fixed length list in a relational database, or as edges in a graph database. Online updates are possible, but convoluted due to cascading cache changes (which can be avoided in an approximate variant by simply not updating existing elements on insertion of new ones).

3.1.2. Network Analysis

As proposed by Son and Kim (2017), it is also possible to find similar items based on their community structure in a graph that reflects all of their attributes (i.e. a multiattribute network). Stronger mutual relationships between real estate properties in such a network suggest a higher degree of similarity. The elements that are most strongly related to a given property in the network could therefore be suggested as comparables. Another approach can be found in Sun et al. (2011), which proposes a similarity metric that uses the number of paths between items in a heterogeneous network along a given metapath (semantic relationship) to identify similar items.

The graphical structure that would be used to represent the properties offers a high degree of flexibility in representation, as categorical features are easier to represent and interconnect this way compared to encoding them metrically. However, relatively little literature exists on the use of networks for content-based recommendation purposes, and examples of implementations are similarly lacking. If we were to apply this method, we would therefore not be able to rely on easily available information, which is a complicating factor. The various features of a real estate property would also need to be accurately weighted for results to be of a high quality. When accurate weights are available, k-Nearest Neighbours might be simpler to implement, as it does not require us to map the problem space to a graph representation.

3.1.3. Infeasible alternatives

Many options that we considered are known to be infeasible in advance, usually due to a lack of essential data or violation of a (non-)functional requirement.

3.1.3.1. Collaborative Filtering

Many popular implementations of recommender systems are built around Collaborative Filtering (CF). Item-based CF involves selecting items that have received a ranking similar to the queried items. User-based CF takes a similar approach from the perspective of users (which is not relevant to our product). The main issue with this approach in our situation is that we do not have the kind of ratings data required for it, and collecting such data would be impractical.

3.1.3.2. Deep learning

State-of-the-art performance is often achieved using deep neural nets, such as in YouTube (Covington et al., 2016). Such solutions, however, require a massive amount of training data to achieve optimal performance, and tend to be based on collaborative filtering (i.e. ratings by similar users). Finally, results generated by deep neural networks are often not interpretable at all. All of these constraints mean that a solution based solely on deep learning would be impractical as well.

3.2. Explainability

One of the requirements is that the end user should be informed, in some way, of how the end result was formed. There are different ways of informing the user/client which factors contributed to choosing a specific comparable. Failing to provide a proper explanation when the results seem unrelated, can result in a user losing trust in the system, thus failing to improve the adoption rate of machine learning models for valuation. The need to explain the result limits which underlying algorithm can be used. It should be possible to generate an explanation from the algorithm.

There are different ways to explain the outcome of an algorithm to the user. These explanations need to be consistent in order for them to increase trust and adoption rate (Smith and Nolan, 2018). In machine learning the human aspect is often overlooked, and when using a machine learning model it is important to know that when a user does not trust the outcome they will not use it (Ribeiro et al., 2016).

In Herlocker et al. (2000) experiments are done where the outcome of a collaborative filtering recommender is explained to users is several different ways. Here it was found that explanations can both significantly increase or decrease the effectiveness of a recommender.

Different approaches to model explanation are:

Feature Impact By showing which features had the largest impact on a result a user can be informed of how a choice was made.

Similarity Level Using a similarity level can help a user understand how good a specific match is. A lower similarity can show that a specific result might not be a good match.

Similarity Level with Silos By grouping features in silos, a similarity can be calculated for each specific silo. When the similarity on one silo is very high the user can take this into account. The silos can be used to give the similarity levels more meaning (e.g. by using a silo for building-related attributes and one for market-related ones), which is less opaque than a compound similarity level because attributes within a silo are more obviously related to each other.

3.3. Software & Architecture

As described in section 2.3, GeoPhy structures its internal code as a streaming microservice architecture. Services send and receive data to/from Apache Kafka topics. They are commonly implemented in either Python or Scala, and need to be capable of handling partial information to maintain an up-to-date view of the data set. This implies certain architectural decisions on our side as well, since the product needs to fit in with existing systems. What remains on our side, then, is to determine how we want to structure our service internally — how will it interact with the data stream, and what language, frameworks & libraries do we use to implement it?

3.3.1. Language

Choosing the best programming language for the job is partly dependent on the algorithm used for comparable selection, as the library ecosystem of the chosen language needs to accommodate for it if we do not want to reinvent the wheel too much. Additionally, it is practical to choose a language that GeoPhy is already using internally, so that the service we develop can be maintained after completion of the project. The second consideration leaves us with two realistic options: Scala and Python.

Benefits of Scala Scala has a powerful static type system. When used properly, this can help avoid certain kinds of runtime problems in our code, and helps cut down on runtime type checking logic and the associated unit tests. Developer tool quality (integrated development environment, static verification, etc.) is also powerful because of it. There is a solid library ecosystem for big data processing and streaming environments. Scala works well with e.g. Hadoop, Spark, Kafka, and Storm, partly because of full Java compatibility.

Scala also has better support for thread parallelism and cluster computing compared to Python (see subsection 3.3.2). Finally, GeoPhy has several libraries available already that can handle the specific message streams in their architecture.

Drawbacks of Scala The Scala machine learning ecosystem lags behind that of Python — libraries exist for many purposes, but are not as popular and/or cutting-edge. However, the libraries that exist do often integrate with existing big data technology better due to Scala's Java-compatibility, and are sufficient in many situations.

Benefits of Python Python has a lot of mind share in the data science community, and many high-quality libraries and tools as a result. While libraries similar to e.g. Scikit-learn, NumPy, or Tensorflow exist for other languages, they are not nearly as popular, and therefore often less mature, less well-documented (in samples, guides & tutorials), and less optimized.

Drawbacks of Python Python has dynamic typing. This makes Python susceptible to runtime type errors, harder to maintain/refactor, and slower. An optional type system is available in the form of mypy, but it is usually necessary to resort to runtime checks and unit tests as well. Because Python is an interpreted dynamic language with a somewhat slow interpreter, pure Python performance does not come close to what is achievable with Scala. However, this is often compensated for by implementing certain logic as a C extension instead, and net performance is usually sufficient. The Python interpreter holds a global lock on its internal objects, and computationally intensive parallelism is not really possible as a result. While I/O operations and C extensions such as NumPy can release the GIL[2], it is still quite problematic. Python developers often avoid threading entirely and distribute work over multiple processes instead, which is less straightforward and harder to coordinate.

3.3.2. Stream processing

While we know where our data will come from, there are still plenty of options for processing that data. Several of these options can also be combined as required.

A relatively straightforward tactic would be to feed relevant data directly into a conventional database (e.g. PostgreSQL). This makes sense for data that needs to be looked up frequently in a shape easily constructed from the stream. Data can also be transformed by another service, fed back into Kafka, and then stored in the database. This is practical if some preprocessing is required. Kafka has the concept of a so-called Connector, which is a service that bridges between Kafka and some other source/sink of data. Connectors already exist for e.g. SQL databases, HDFS[3], and stream processing clusters. The development workload and risk associated with this tactic is therefore very low.

Another possible approach is to structure (part of) our service as a Kafka Stream. A Stream is fed data from Kafka topics, and outputs data to Kafka topics as well. It can therefore serve as a transformation step, and used for data pre- or postprocessing as well as aggregation. Streams are managed by Kafka itself, and help avoid the need for a separate stream processing framework. They are scalable and fault-tolerant, and can be developed in plain Java/Scala[4]. The programming model is simple, which makes it easy to use and requires very little infrastructure. However, it can be slower when intensive calculations are required, and lacks some of the more powerful primitives available in dedicated frameworks.

It is also possible to use a dedicated stream-processing framework to interact with Kafka. Examples of these are Apache Storm, Apache Flink, Apache Spark Streaming, and Apache Samza. These frameworks differentiate themselves on e.g. micro-batching vs true streaming, use of YARN[5] vs custom scheduling, and level of abstraction.

Finally, Kafka also exposes simple(r) consumer & producer interfaces, which can be used to simply publish and/or subscribe to a topic and build arbitrary code around it. The persistent nature of Kafka makes fault tolerance relatively simple to ensure, so simple tasks that do not benefit from large-scale parallelism within a consumer can be implemented this way. Producer/consumer support is also the level of functionality that is exposed by non-Java client libraries (e.g. Python).

[2] *Global Interpreter Lock*
[3] *Hadoop Distributed File System*, see `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`
[4] An unsupported Python implementation also exists
[5] *Yet Another Resource Negotiator*, Hadoop's native scheduler and resource manager. See `https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`

3.3.3. Machine Learning

At the core of our service is the ability to learn and subsequently apply a method for identifying comparable properties. On an algorithmic level, several possible solutions to this were mentioned in section 3.1. Each of those algorithms, however, can be implemented in different ways. Below are a variety of options for the execution of algorithms in several machine learning categories. They range from manual implementation to prebuilt high-level frameworks. Note that auto-encoders are technically neural networks, and would most likely be implemented using one of the options in subsubsection 3.3.3.4 instead.

3.3.3.1. Pre-processing

Apache Spark offers various row-wise and aggregate computation primitives over dataframes (i.e. tabular data) in Java/Spark/Python/R, which can be used to implement efficient parallelized preprocessing logic. It is also possible to layer **Spark MLlib** over this, which provides implementations of popular preprocessing steps, as well as a pipeline abstraction that can be used to split and combine various such steps. At small scales, it tends to introduce significant overhead, but for larger data sets, the parallelism can outweigh it.

Apache Flink is a framework for streaming data processing at scale. It offers abstractions at multiple levels for both tabular data and streaming records, which can be further combined at will. In addition, it has the **FlinkML** library, with primitives for certain preprocessing steps and a pipeline abstraction not unlike that of Spark MLlib.

Pandas is a Python-specific library that offers dataframes much like those in Spark, but only in-memory on the local machine. While not quite as scalable, it is very intuitive and is fast enough at smaller data set sizes.

Scikit-learn is a Python library that contains many implementations of algorithms related to machine learning, as well as an abstraction for pipelining such algorithms.

Manual implementation of the preprocessing logic is also possible, and usually not particularly difficult. However, it is more practical to only fall back to this if a specific algorithm is missing from one of the above solutions.

3.3.3.2. k-Nearest Neighbours (see subsection 3.1.1)

PostgreSQL has the `cube` datatype, which can represent points in n-dimensional space. Feature coordinates (especially when dimensions are relatively few) could be stored in such a column and indexed, after which a LIMIT'ed query ordered by distance from a reference point could be executed with help of this index structure. Performance needs to be evaluated, especially if we were to use multiple different coordinate systems for silo'ed similarity levels and an aggregate distance minimization would be required. In the worst case, such queries could resort to comparing every point pair distance, which might still be feasible but would most likely cost seconds at full database load.

Scikit-learn contains k-NN implementations, each of which internally builds a tree structure at training time. Those tree structures could then be stored and retrieved. One problem with this is the apparent lack of online updating that the sklearn implementations suffer from. Rebuilding the structure whenever a new item is inserted is most likely infeasible, although updates could be batched for e.g. daily tree restructuring. Performance is also likely to be limited, due to Python's interpreter speed and non-parallelism.

Apache Spark can be combined with the **spark-knn** library, which is able to construct a hybrid spill tree (see subsubsection 3.1.1.3) at training time for k-NN lookups that can be stored and retrieved. Due to its use of Spark, it is capable of parallelizing tree construction to some extent. However, online tree updates are once again not an option, and feasibility is therefore constrained by duration of full tree construction.

Turi (formerly GraphLab) is a graph-based Python library for distributed machine learning. Turi contains an acceleration structure for nearest neighbor queries, including support for weighted silo'ed distance calculation. It also supports pre-processing, and can optionally integrate with MXNet (see subsubsection 3.3.3.4) for further deep learning support.

nmslib is a toolkit for similarity search that provides various metrics, acceleration structures, and other functionality useful in this context. It achieves high performance according to Naidan et al. (2015).

Manual A custom implementation of a tree structure is also possible — see the paragraph on k-NN search in subsection 3.1.1 for some of the options. The custom structure can be customized for optimal behavior and performance characteristics in our specific situation, but implementing it obviously involves more work than using an existing library.

3.3.3.3. Network analysis (see subsection 3.1.2)

NetworkX is a Python package containing data structures and algorithms for complex networks. It is widely used, and has one of the most extensive collections of algorithms available in a single library. However, it is limited to in-memory graphs on a single machine.

GraphX is a Spark toolkit for graphs and graph-parallel computation. By utilizing Spark, it supports large scale parallelism and data distribution.

Turi specifically supports item content-based recommender systems that operate on graphs. It is based on k-Nearest Neighbors internally.

Neo4j is a popular graph database with a flexible query language and implementations of several high-level graph algorithms. It is highly scalable, and well-supported database drivers exist for both Java/Scala and Python. A large advantage of picking it over an ordinary graph library is that it is an actual persistent database. Storing a built network is possible for the other libraries, but the network has to fit in memory and the storage has to be fault-tolerant or the network reconstructible. Neo4j takes care of this itself.

OrientDB and ArangoDB are multi-model databases with native support for graph primitives. The multi-model nature means that it is not required to force our data model into a graph — it is simply possible to utilize e.g. document storage instead where this makes sense. This comes at the cost of domain-specificity — Neo4j is more focused on graphs.

3.3.3.4. Neural networks & deep learning

Tensorflow/MXNet/CNTK/Caffe2/PyTorch are computation graph-based tensor processing libraries that operate at a relatively low level. They expose the primitives required for stacking and distributing differentiable computations on n-dimensional arrays (i.e. tensors), as well as implementations of higher-level constructs (e.g. dense neural network layers). Between them, there is a distinction between a define-by-run and a define-and-run approach: either the computation is defined imperatively and runs on definition, or it is defined symbolically in advance and submitted for execution afterwards. Symbolic execution allows for more optimization in theory and is easier to implement in some ways, but allocates all memory upfront and is generally harder to introspect, debug, and extend (Tokui et al., 2015). The various libraries further distinguish themselves on execution speed, community strength, and distributed execution support/method.

Keras/Gluon are neural network libraries that operate at a higher level of abstraction than the aforementioned libraries, but use them under the hood. Designing a model mostly involves composing high-level layers instead of mathematical operations over tensors, although the lines can be blurry due to higher-level abstractions present in e.g. Tensorflow. Both Keras and Gluon are agnostic to the underlying tensor processing framework. Keras has better support for multiple libraries (Gluon is limited to MXNet for now) and has a larger community.

Deeplearning4j is a neural network library that distinguishes itself by supporting Java-compatible languages and the surrounding ecosystem. It has support for state of the art models and accelerated execution like the above libraries, but in addition also supports e.g. Spark for distributed training. Finally, Deeplearning4j is capable of importing Keras models, which allows developers/data scientists to build their models in Python using a high level interface, and deploy them in Java/Scala production environments.

4 Architecture Design

This chapter was written at the outset of our project, and has been left in its original state. The actual implementation has evolved in response to what we learned during development. Any conflicting information in chapter 5 therefore supersedes what is stated below.

In the previous chapter, we stated several possible approaches and techniques we could apply for developing the perfect comps product. In this chapter, we state the choices that we have made in order to build it. These choices are supported by the information stated in the previous chapter and the arguments given in this chapter. They are also evaluated in the context of all requirements stated in section 2.4.

4.1. Comparable property selection

Comparable selection will make use of nearest neighbor search, as described in subsection 3.1.1. k-NN is a natural match for our problem, and a versatile solution because of the many different methods that can be used to encode a property as a coordinate. It does not constrain the structure of our data, can be accelerated well, and is simple to maintain even with a rapidly evolving data set. Network analysis-based methods would have required more complicated modeling of properties into a suitable structure, and not yield any benefits that are obvious upfront.

4.1.1. Pre-processing

For the pre-processing logic, we will start with a naive approach, so that we can iterate rapidly and have a working prototype quickly. The pre-processing will be built with the option to integrate different kinds of dimensionality reduction, like PCA, SVD and Auto-encoders. We will need to find out what kind of dimensionality reductions will work best for the data that is processed by the product. Features will be weighed according to their relative importance to comparable selection. Finding these weights is a challenge in itself, as there is no obvious ground truth to derive it from. We will attempt to find them using a combination of two methods: extraction from a set of existing human-generated comparable sets (which is likely biased, but can be valuable to identify importance of less biased features) and from feature importance in predicting values that could be a good proxy for similarity (e.g. capitalization rate, which is sometimes used by human appraisers to value properties). We might also take advantage of the weights used by the AVM.

4.1.2. Distance metric

Most comparison implementations contain a default set of distance metrics — often at least Euclidean, and optionally Manhattan/Chebyshev. We will limit ourselves to these unless they prove insufficient, and experimentally determine which one is most effective.

4.1.3. Comparison

For comparison/search, a service will be built which abstracts over a tree-based k-NN index. In this service we will implement a search tree that is constructed from the preprocessed data. The abstracted service will allow us to implement different kinds of trees, as there does not seem to be one specific implementation from the offset that is a perfect fit for our problem. We will prioritize testing of trees with support for online updates.

4.2. Explainability

In order to achieve explainability, we will keep the need for it in the back of our heads while implementing and choosing an algorithm. The algorithm should in some way be able to expose the distance of each feature in the comparables.

Once the algorithm achieves sufficient accuracy, we will use the distances of the features to calculate a similarity score in a specific range. We will need to analyze the results to find a similarity score that makes sense in our specific domain. This will most likely utilize a method that consists of scoring features in silos. These silos will represent different aspects of the similarity. GeoPhy has proposed to use property, location and market. We will investigate whether these silos are the most sensible choice.

4.3. Software & Architecture

As was mentioned in subsection 4.1.1, there are many open questions on pre-processing, and on an algorithmic level in general. The only way to answer these is through a process of continuous prototyping. In contrast, it is much simpler to predict in advance what the impact of software architecture decisions will be. Keeping this in mind, our choice of programming language has fallen on Python for at least the model selection phase of development. The size of the data science ecosystem around it ensures the availability of high-quality libraries and tools that are very suitable for rapid iteration, and will help us answer the open questions while avoiding implementation details. During subsequent integration into GeoPhy's existing system, we will evaluate whether it is worth porting over (components of) the system to Scala instead to improve cohesion.

This does not mean that we will ignore the final implementation during experimentation, of course — whatever we build during this phase will have to be adaptable to a scalable solution in the end. As mentioned above, we might want to port the implementation to Scala, but even if we do not, it is still important that the end result is maintainable and fits into GeoPhy's full system. Anything implemented from scratch should be usable in or migratable to the final product with minimal effort. It therefore makes sense to implement any interaction between components using the same abstracted interfaces we intend to use in the final solution. This could prove useful during prototyping as well, as it allows us to swap out different implementations of e.g. pre-processing or nearest neighbor search and compare behaviour.

4.3.1. Components

See Figure 4.1 for an overview of the components.

Combination service Combines messages from Kafka topics into denormalized entities. The entities are persisted in a data store for future reference and updates.

Pre-processing service Transforms entities from the combination service to low-dimensional vectors. This is achieved through feature engineering and dimensionality reduction. Generated feature vectors are either published to a Kafka topic or directly submitted to the comparison service, depending on which one turns out more convenient. The vectors can be interpreted as points in a multidimensional space that can be used for k-NN.

Comparison service Ingests feature vectors generated by the pre-processing service, and exposes a way to request a list of comparables for an input property that has already been ingested by GeoPhy infrastructure. Generating a comparable for a previously unseen property involves ingesting it into the core infrastructure beforehand. Internally, the service will maintain the acceleration data structure as described in subsection 4.1.3, and query it on request.

The separate services are relatively independent. The combination service does not know about anything but assembling entities. The pre-processing service does not know where entities come from, just that they exist. The comparison service does not need to know about the pre-processing service at all, as long as it receives feature vectors that are part of a metric space. The pre-processing service knows it needs to generate this,

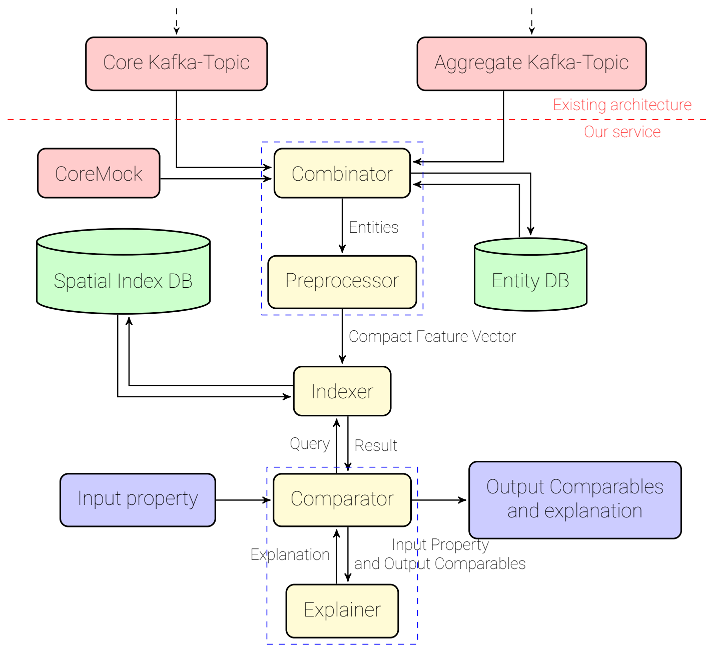

Figure 4.1: High-level overview of the architecture

but nothing about what will happen with the end result. All services can (largely as a consequence of this) be swapped at will for another implementation, as long as the interface stays the same. The pre-processing and comparison services will be slightly codependent in the sense that either one assumes metric learning-compatible behavior of the other, but this can not realistically be avoided without merging them. Separating them does offer value, as resources and failure for data store aggregation and pre-processing are separated from those of maintaining and querying comparables.

4.3.2. Stream Processing

Newly added data points will need to be combined into entities in order to be able to execute the pre-processing logic. New data points can originate from both the core and aggregate databases, and it is not guaranteed that all data is published at the same time. Retaining some state in between processing messages from the streams is therefore required, as is the ability to merge streams.

Apache Flink has features that allow for such combination logic very easily: it connects directly to Kafka, after which you can create unioned streams from multiple topics (in our case, core and aggregate), and then applying a keyed window function to buffer all messages that concern the same entity and follow each other closely. The collected messages can then be inserted/updated in a database, after which the combined entity can be retrieved and run through the pre-processing logic. Flink is already in use within GeoPhy, can be used from both Python and Scala, and is highly efficient. The Kafka compatibility and stream processing primitives it offers will simplify implementation compared to custom processing, and more natively fit the architecture than e.g. Spark Streaming.

Property data will be persisted in PostgreSQL, which is already used in other GeoPhy services and offers the data model flexibility we need. The database schema is set up in such a way that non-essential data can be missing - a property's defining attributes need to be present, but additional information is stored as a loosely structured document instead to allow for sparsity in storage. It can then be compensated for at pre-processing time.

4.3.3. Machine Learning

As mentioned in subsection 4.1.3, comparables will be found by encoding them as coordinates in a metric space and running a nearest neighbor search, accelerated using a tree structure. The pre-processing logic will be iteratively developed (simplest method first) in Python during initial prototyping. We will use Scikit-learn to develop a suitable pipeline of its built-in pre-processing logic, optionally amended with custom Python stages and, if auto-encoders become relevant, Keras, which integrates well with Scikit-learn pipelines and offers a high-level development experience. If we decide to port this to Scala, Scikit-learn has an analog in FlinkML (which supports a similar pipeline abstraction and stages), and any Keras models can be implemented as-is into Deeplearning4j. Alternatively, the Python pre-processing logic can be encapsulated so that it can be used as a Flink transformation from Scala.

PostgreSQL will be used for nearest neighbor search. The preprocessor-generated coordinates of each property will be stored in a column of type `cube`, which can store n-dimensional points, calculate distances between them, and build an index for accelerated nearest neighbor search among other things. PostgreSQL takes care of online updates to the index tree as well as disk persistence of it, and use of an index makes it sufficiently performant for our use case. An additional advantage is that interaction with PostgreSQL is language-agnostic, and use of Python or Scala does not actually matter.

5 Software Implementation & Testing

In this chapter, we evaluate the implementation process of the architecture described in chapter 4. We also describe the final outcome from a software perspective, as well as any architectural changes we made along the way. Our service was developed using a loosely agile process, and its components therefore evolved gradually from being minimal to fairly complete. This is reflected in the structure of this chapter — the evolution of our system is described sprint-by-sprint for each of its conceptual components.

At the start of the project, we intended to first tune a model and to later build the service architecture around it. Fairly early during the implementation process, however, we decided to approach this in reverse order: set up the architectural scaffolding early on with a very simple model inside, and gradually improve the model (as well as the architectural robustness and readiness for integration) from that baseline. This approach felt more natural to our inner software engineers, and better facilitated collaboration by providing a clear structure to improve upon. It also facilitates ongoing model improvement without major adjustments to the architecture as new data and models are explored.

5.1. Comparable property selection

5.1.1. Sprint 1

Our focus during the first sprint was to develop the baseline implementation of our architecture. Model performance did not factor into this, and was therefore pushed forward to the later sprints by substituting a dummy preprocessor for the time being. The dummy preprocessor acted like any real implementation would: given the data of a complete real estate property, it returned a low-dimensional vector that could be used in a k-Nearest Neighbors algorithm. However, it did this by simply returning two fields of the property as-is, which kept things simple but not particularly accurate.

The k-NN implementation as it ended up in the final product was implemented during this sprint, and did not change from an algorithmic perspective afterwards.

5.1.2. Sprint 2

With the initial architecture in place, some of our focus could be switched over to model development. The dummy model from sprint 1 provided the structure that the final model would need to adhere to, but many details still needed to be fleshed out, including a mechanism for persisting model parameters and one for experimenting with different models. Python's built-in serialization framework, pickle, was a promising option for this, as it is capable of serializing almost anything that's thrown at it. However, it had some issues with more complicated data structures (such as Scikit-learn models), which we worked around by using dill[1] (McKerns et al., 2012) — an extension of pickle that provides even more flexible serialization.

Building this flow was fairly effortless, and model prototyping could be done in Jupyter[2] notebooks as a result, which facilitated effortless prototyping. The dummy model was replaced by a version that picked several property attributes that were expected to correlate well with similarity, normalized them to zero mean and unit variance (standard normal distribution), and weighed them by an importance factor.

[1] See `https://pypi.org/project/dill/`

[2] See `http://jupyter.org/`

5.1.3. Sprint 3

In order to determine some more informed weights for our simple weighted normalization model, we used the optimization library hyperopt[3] to evaluate the weight search space and come up with a combination of them that minimized an error function. Geophy's domain experts suggested that, while not an exact ground truth, the so-called capitalization rate[4] would be a good proxy to assess comparability, and this information was used to develop a suitable error function. It constitutes the mean squared difference between the capitalization rate of a sample of subject properties and each of their comparables.

Collaboration with a data scientist on the team also resulted in a more complex model, based on a Kernel PCA, that could be swapped in. This model uses kernel functions to project the entire 700-dimensional feature space of a property onto a higher-dimensional linear feature space, of which it then calculates the principal components.

5.1.4. Outcome

As was the idea from the design phase onward, the core model logic of the Perfect Comps service resides in the preprocessor, which is fully swappable and can be improved upon without too much interaction with the system's other components. Above a certain level of sophistication, preprocessor models often require a training phase to tune their parameters for the input data set, which is captured in our service by serializing the model in its entirety (including parameters and code). The service's core logic merely specifies an interface that a preprocessor needs to adhere to, and some logic for loading one from a highly flexible serialization format, the aforementioned dill. Models can then be trained in e.g. a Jupyter notebook with a set of offline data, wrapped into an implementation of the preprocessor interface, and stored to disk — code and parameters included.

There are currently two tuned models that can be swapped into the service without too much effort. First of all, there's the simple normalized & weighted model, for which we hand-picked several features from the data set that either intuitively made sense to use, or that showed high correlation with indicators such as capitalization rate or transaction price. The process of selecting these features and weights involved many iterations of adding/removing a feature, optimizing weights for minimal error on one or more performance indicators, and evaluating performance across a large fraction of the data set. Performance of the best model to date (on the capitalization rate) is a MAPE[5] of 11.8% and MdAPE[6] of 8.8% with a standard deviation of 11.8 percentage points. The second model is a Kernel PCA over all features in the data set (without weights), with its parameters tuned for minimal reconstruction error of the original features. This model achieves a MAPE of 13.8% and a MdAPE of 9.8% with a standard deviation of 25.5 percentage points, which is worse than the simple model (especially the standard deviation), but would likely improve if features were weighted by relative importance. Additionally, its implementation can be made less sensitive to missing data points, which would be hard to achieve in the simple model. As a frame of reference, we also determined the metrics that resulted from a model that randomly selects 10 comparables from the data set. These turned out to be a MAPE of 18.5%, an MdAPE of 14.0%, and a standard deviation of 27.5 percentage points. Both models therefore perform significantly better than random selection.

The k-NN search makes use of the PostgreSQL cube datatype (as described in subsubsection 3.3.3.2). The feature vectors as produced by the preprocessor are stored in a two-column database table, mapping the property's unique identifier (a string) to its accompanying feature vector (a cube with volume 0, representing a point). A PostgreSQL GiST [7] index is present on the feature vector column and provides the necessary accelerated search operations. k-NN search is executed using a simple SQL query that, when provided with the identifier of a subject property, selects the top n comparable property URIs through sorting by their feature vectors' distance to that of the subject. The sort operation utilizes the GiST index and carries the cost of only a few tree lookups. Multiple distance metrics (as described in subsubsection 3.1.1.2) are supported (Euclidean, Taxicab, and Chebyshev) through changing only a single operator in the query.

[3] Distributed Asynchronous Hyper-parameter Optimization, see `https://github.com/hyperopt/hyperopt`

[4] Net operating income divided by transaction price

[5] *Mean absolute percentage error*

[6] *Median absolute percentage error*

[7] *Generalized Search Tree*, see `https://www.postgresql.org/docs/current/static/gist-intro.html`

5.2. Explainability

5.2.1. Sprint 1

Early on when discussing our architecture, we decided that explanations would most likely need some statistics on the composition of our data set and each of its attributes. This was achieved by adding some logic to the combinator. For every ingested property, it updated rows in a PostgreSQL table corresponding to each of the property's populated attributes. While limited to the minimum and maximum values during this sprint, it later proved easy to extend this mechanism with further information.

Due to a focus on bringing up a functional prototype during this sprint, very little attention was paid to the explanation metrics just yet. Results from the service included a minimum and maximum for a hand-selected subset of each comparable's numeric attributes, but no meaning was attached to it yet. However, this bare minimum of functionality did show that the statistics were properly being maintained by the combinator.

5.2.2. Sprint 2

To provide some form of comparison metric in the service's output, z-scoring[8] was implemented during this sprint. The statistics table was extended with three new fields to support this: the number of values that have been seen, the sum of those values, and the sum of squares of those values. This allows us to track the standard deviation of each attribute's set of values X continuously as new data streams in on a stream:

$$\begin{aligned}
count_X &= \sum_{x \in X} 1 \\
sum_X &= \sum_{x \in X} x \\
sqsum_X &= \sum_{x \in X} x^2 \\
stddev_X &= \sqrt{\frac{sqsum_X}{count_X} - \left(\frac{sum_X}{count_X}\right)^2}
\end{aligned}$$

The z-score is less sensitive to a large value spread on an attribute's domain (which is dictated completely by even a single outlier), and therefore closer to what a proper similarity metric would have to look like. From a UX perspective, however, it is still too mathematical for most of the intended user group.

5.2.3. Sprint 3

To provide a more intuitive measure of similarity, the z-score was transformed into a score between 0 and 100. This turned out to cut down similarity in those cases where the absolute difference in value of an attribute was very low, which was compensated for by also mixing in the absolute difference of the compared values compared to the total spread of the attribute.

Finally, the number of features that are available for most properties make it impractical for a user to assess similarity between comparables at a glance. Instead, we added summary metrics, based on silos of attributes in the original data set and an aggregate of their individual similarities.

5.2.4. Outcome

Explanation of the generated comparables' similarity to the subject property has been made orthogonal to the comparable selection itself in our service, in order to avoid restrictions on and by model structure. By keeping track of several statistics on data set composition at ingestion time, information is available to the explainer that it can use to assess the closeness of each individual feature. To provide a more compact overview of overall property similarity, these feature closeness scores are then summarized in a few meaningful silos (composed of a subset of attributes), each of which receives an overall similarity score that is the weighted average of its components. There are three silos presented in the product: property, location, and market.

[8]A measure of similarity with as its unit the number of standard deviations in between

Given a feature f, a subject x, and a comparable y:

$$
\begin{aligned}
zscore_{x,y} &= \frac{f_y - f_x}{\sigma_f} \\
spread_{x,y} &= \frac{|f_y - f_x|}{f_{max} - f_{min}} \\
c_z &= 0.5 \\
p_z &= 0.6 \\
c_{spread} &= 0.5 \\
similarity_{x,y} &= c_z * p_z^{|zscore_{x,y}|} + c_{spread} * spread_{x,y}
\end{aligned}
$$

Various parameters of this formula can be tuned to achieve e.g. maximally accurate or visually pleasing similarity score distributions. For example, while the z-score metric alone would provide a less ambiguous metric on its own, mixing in the spread ensures that, when features have a small and insignificant spread, the scores end up higher on average, which could be more satisfying from a UX perspective. Which one of those is more important is a question that remains to be answered through e.g. user testing (see section 8.4).

5.3. Stream Processing

While the intention was initially to use Apache Flink as a stream processing framework, we discovered early on that it would be difficult due to the way Flink exposes its Python integration — the library was restricted to Python 2.7, and bundled in a nonstandard way. In addition, our stream processing logic didn't benefit that much from the unique selling points of Flink, and was simple enough that it could be handled by some custom code instead.

We decided to express our stream processing logic as an abstraction that forwards batches of partial property updates (grouped by transaction) to a callback object. The callback logic then handles steps such as update merging, preprocessing, statistics tracking, and index insertions. The stream consumption abstraction has been implemented with time window-based batching and fault tolerance in mind. Messages are batched by transaction until no new ones arrive for some seconds, after which they are handed off to the callback object. All throughout this process, they stay marked as pending, so that if any exception is raised, the input stream can be wound back to the last message known to have been processed. That, combined with Kafka's persistent messages that are tracked by offsets per consumer (and partition), allow for some level of fault-tolerance — as long as Python is able to still catch exceptions and reach Kafka, no message goes unprocessed. If needed, this can be extended for even better fault-tolerance by switching to explicit offset commits instead, which would also catch those rare cases where either Kafka is unreachable or Python can't execute its exception handler.

5.4. Service Architecture

5.4.1. Sprint 1

Bringing up the architecture as described in chapter 4 was one of our priorities during this sprint. The statistics store was added to the preprocessor early on to facilitate explanations.

5.4.2. Sprint 2

After reviewing the outcome of sprint 1, we decided to switch indexer insertion from Kafka to HTTP. By doing so, the access patterns of the indexer became more consistent, and better prepared for scaling.

5.4.3. Sprint 3

Due to reasons expanded upon in section 5.5, we switched over our entity data store from PostgreSQL to an RDF triple store.

5.4.4. Outcome

Our overall architecture has largely stayed consistent with what was described in chapter 4 at a component level. At the interface level, however, a few changes were made to better reflect the usage patterns of each service.

First of all, the connection between the preprocessor and the indexer has become HTTP (as opposed to Kafka). The pull-based access pattern of Kafka would have forced the indexer towards full resource utilization whenever many new feature vectors arrive. This would impact the individual scalability of the service, as it could become unresponsive to requests from the comparator during such phases. Switching to HTTP avoids this problem, as the indexer now only exposes a request-based interface that is equally likely to answer to the preprocessor or the comparator. The switch also increases internal consistency, as the indexer only has a single interface protocol now. It is still possible to add a (Kafka-based) buffer service between the preprocessor and indexer as a load-balancing measure — see section 8.1 for more on this.

Secondly, we switched to an RDF database as our entity data store. This facilitated the incremental patterns of upstream data streams better. More information on this can be found in the next section.

Finally, a statistics data store was added (in response to what was described in section 5.2) in the form of PostgreSQL. For each property attribute, it tracks several statistics that can be used to generate explanations. The current implementation does have some scalability-related drawbacks, due to its reliance on a small number of rows (a few thousand at most) being updated frequently. PostgreSQL locks its rows during updates, and this locking could become a bottleneck over time, although it is hard to say how soon. More on this in section 8.1.

5.5. Integration

Integration into GeoPhy's infrastructure allows our service to seamlessly work with all the information available in the system. This does, however, create some new challenges. Most of the non-functional requirements are the result of integration into the GeoPhy infrastructure.

5.5.1. Sprint 1

The first step towards integration was to make sure that the preprocessor could consume messages from Kafka and process them. This was done without taking into account what the actual data structure looks like. This version split a CSV file into data points for each cell, which was then published onto a Kafka topic. The preprocessor consumed the topic and recreated the rows from the CSV file. Each recreated row was stored inside a PostgreSQL database in a JSON field for further reference.

This allowed us to develop the preprocessor in a streaming fashion without having to implement the same data structures that were already used in the GeoPhy infrastructure. It enabled us to create a first prototype very rapidly.

5.5.2. Sprint 2

On the part of integration, this sprint consisted of refactoring and improving the existing code.

5.5.3. Sprint 3

During this sprint, the system that our service integrates into was presented, and version 1.0 was released. This allowed us to focus more on integration, as the infrastructure was not subject to large changes anymore. The preprocessor and coremock modules were rewritten to take into account the nested structure of the data they receive.

The PostgreSQL combination logic was adapted to use an RDF database instead. This allowed us to not make any assumptions about the incoming data, and also to store incomplete data without a need for complex logic that keeps track of partial entities.

5.5.4. Outcome

When starting out, it was not completely clear how integration with the GeoPhy infrastructure would work, as the infrastructure was not yet finished when development on Perfect Comps started. This is why the choice was made to assume a simplified version of the expected data, so that we could iterate quickly.

In the final version, any incoming data is immediately stored inside an RDF database. This store is queried using SPARQL[9] to create a flat data structure that describes properties from the nested data. Only properties for which sufficient data is available are processed. When data about a property is still incomplete, the data is still stored in the database, so that when new data arrives that completes the property, it is run through the preprocessing logic in full.

[9] *SPARQL Protocol and RDF Query Language*

With this solution in place, the implementation is completely independent of the order in which new data arrives. New data that invalidates old data can also be dealt with easily. When scaling this solution or running multiple different versions, a drawback is that every service needs to maintain a copy of the database, which is very inefficient. More on this in section 8.1.

5.6. Code Quality & Testing

Ensuring that our code would be maintainable and behave correctly was an important priority throughout the project. To be able to objectively check and enforce this, we took several process measures from the very start. This has helped us to both keep our code clean & consistent, and to prevent many errors from popping up after adding or refactoring something. Especially in a dynamically typed and flexible language like Python, having such practices in place are key in order to end up with a code base that is future development-proof.

By utilizing Git with a strict pull-request policy and leveraging Continuous Integration (CI), we forced ourselves to consider code quality at all times. The CI system automatically runs tests and code quality checks before code is merged into the master branch. Code is also only allowed to be merged into master when both CI and all team members agree (i.e. any feedback has been dealt with).

To keep code readable and understandable, we used several open source tools from the Python development ecosystem. Black[10] was used to enforce highly consistent code style. With the audit tool Pylama[11], our code was checked for unused code, function complexity, common Python pitfalls, and several other types of inconsistency. Finally, most of our code contains type annotations checked with mypy[12]. This borrows some of the good parts from statically typed languages that help make software less susceptible to type-related errors.

Automated testing is one of the most consistent ways to ensure correct behavior — if done right. Using unit tests, our code has been thoroughly tested to make sure all different components work as they should. Additionally, we developed a smoke test and an end-to-end test (run as a final step during continuous integration) to help protect us from regressions in runnability and external interface correctness respectively. The smoke test checks that the service is able to start and run for a while without crashing; the end-to-end test ingests a small dataset and checks that all of its properties are assigned their expected comparables. This exercises all of the public interfaces of the service, as well as all of its internal subcomponents.

Finally, we engaged in frequent manual testing to check the characteristics of our service that are not as easy to encode in test logic. This included the user interface, as well as the high-level behavior of the model itself. Due to its inexact nature and ties to (currently) intangible user knowledge, this is hard to capture in a test that is helpful).

[10]The uncompromising code formatter, see `https://black.readthedocs.io/en/stable/`
[11]Code audit tool for Python, see `https://github.com/klen/pylama`
[12]Optional static type checking for Python, see `http://mypy-lang.org/`

6 Evaluation

In this chapter, we will evaluate our project from process to outcome to impact. Much has happened over the duration of the project, both in terms of our final product and of the development process itself. We will reflect on things that went right, and things that could have been done better. We will motivate our fulfillment of the requirements we listed in section 2.4, and touch upon the impact that our service will have from this point forward — both from the perspective GeoPhy, and from the broader view of the real estate industry.

6.1. Final product

6.1.1. Functionality

At the outset of the project, we determined one of the functional requirements to be that the comparables had to show maximized similarity. Throughout the development process, during interaction with stakeholders inside GeoPhy, it became clear that this should have been stated in less absolute terms. Due to imperfections in the current appraisal process and the intangible aspects of comparable selection, maximal similarity is relative, and varies by user & use case. It therefore needs to achieve good enough performance, but not necessarily optimal similarity. In addition, the outcome of our project was to be a proof of concept, and model optimization was less important than demonstrating viability through at least a working model and accompanying service.

We believe we have succeeded at this. While model output can still be improved in order to be more robust towards outliers, and user tests might be required to steer towards more informed metrics, it shows promise even at this stage (see section 8.4 for ideas on how to move towards production-readiness). A customer deliverable during our third sprint demonstrated this — the output of our service measured up against hand-picked comparables on important indicators , and was generated in a much smaller amount of time than hand-picking them would have taken.

Preprocessing logic is highly flexible, and capable of taking into account relative importance of certain features. We achieved some of our best model performance (measured by capitalization rate error) using a very simple model that merely normalized and weighted a small number of features, and though these weights might be tunable for even better performance, the process of doing so is trivial due to our service architecture. Proper weight selection steers the model into the direction of similar feature characteristics as a result, and important property characteristics such as construction year, number of units, and area similarity are captured by this mechanism. The output of our final model produces comparables that show similarity on such attributes, and that are placed in similar settings (e.g. suburban subject properties are matched with suburban comparable properties), without needing any hard-coded constraints to do so. Our explanatory similarity scores appear to reflect this, and can be tuned in response to user priorities to provide a better experience.

There are, of course, aspects that could have been done better. To give an example: exploration of more complicated models has not been as thorough as we would have liked. This has been due to early performance wins using simple models, time constraints, small data, and our personal experience bias towards software engineering as opposed to data science. Our focus on simple models has helped us deliver a more complete proof of concept. We do think this has been a fair tradeoff, but some of the avenues of exploration mentioned in chapter 3, such as auto-encoders, still lie open and might be interesting to pursue in the future.

Additionally, the relatively small data set we had access to during development (about 7000 sparsely populated rows) makes it hard to assess how well the current model, as well as the architecture in full, will generalize and scale. While we instinctively expect that more data would improve the quality of comparables, it is also quite possible that e.g. feature weights are fit to the current set of properties too much — only data will tell. In section 8.2, some recommendations can be found on how to progress from this point, taking into account the aforementioned evaluation.

6.1.2. Architecture

The fact that our service would end up in a streaming and incremental architecture was known from the outset. Because of that, our every design decision was made with this constraint in mind. We developed a mock version of GeoPhy's platform (which was still under development at that point) during the first few weeks and made sure to develop nothing that would not be compatible with streaming data in general. The lack of a functional reference implementation, and assumptions that we made without a full overview of how that would eventually work, did push us down the wrong path for a while on this subject, but due to several layers of abstraction, the resulting fix fit into our architecture without too many problems (see section 5.5).

Sparse and temporarily incomplete data is dealt with effortlessly by our database, as RDF triple stores lend themselves well to that. On the model side, the story is somewhat more complicated and may change down the line depending on the data set. Currently, data is processed only if all crucial attributes are present, and ignored (though cached for later re-evaluation) if not. Which attributes are critical depends on what model is being used. Some of the simpler models we prototyped depended on receiving complete data for a small subset of attributes. This allows very sparse data to still be ingested, but requires the critical attributes to be well-populated throughout the data set. Models that use and compress the entire feature space, on the other hand, are influenced less by missing features, and imputing them using a simple method (e.g. filling them with the mean value of all entries that are populated) becomes a possibility. Finally, there are some avenues of exploration left that could be capable of imputing data based on the attributes of somewhat similar properties.

Our services have been engineered for scalability, as well as fairly low ingestion latency. Local testing suggests that this has succeeded in the sense that multiple instances of system components can be run in parallel, and that data updates are processed at a high rate. The complete ingestion workload can be evenly split among a horizontally scaled number of instances to further improve throughput & latency. It remains to be seen where the bottlenecks are situated when scaling this up to larger numbers of nodes, but no obvious dependencies exist among the services themselves that prevent a large-scale deployment. Scale is always relative, of course, and the presence of e.g. a relational database would most likely prove limiting when dealing with truly big data. However, at the scale that GeoPhy is currently dealing with, this should not form a problem yet, and if certain components of our system do turn out to be limiting, they can be replaced individually to achieve better performance (see section 8.1 for some suggestions on how this might be done).

6.1.3. Code base

One of the first decisions we had to make at the start of the project was whether to use Python or Scala to implement our service (see subsection 3.3.1). We decided to (at least initially) develop everything in Python. This was due to the intuition that its flexibility and large data science ecosystem would help us iterate more quickly than Scala would have.

While it is hard to say whether Scala would have been a better or a worse choice in hindsight (after all, we did not use it), we feel it is safe to say that Python was a solid choice, and that its benefits have outweighed its drawbacks so far. In part due to a principled and abstracted architecture, and due to the extensive use of code quality checkers and enforced type annotations, our code base has not felt fragile or hard to refactor, and changes could be made without having to fear bugs or regressions.

The data science aspects of the project were made significantly easier by high-quality libraries such as Pandas and Scikit-learn. Due to Python's (and dill's) flexibility in object (de)serialization, that part of the code is well-separated from the more infrastructural parts. Recent advancements in the language around asynchronous processing[1] have elegantly alleviated some of Python's issues around concurrency, and many

[1]Introduced in `https://www.python.org/dev/peps/pep-3156/` (2012) and `https://www.python.org/dev/peps/pep-0492/` (2015)

high-quality libraries (such as aiohttp[2], aiokafka[3], and asyncpg[4]) exist that enable the development of non-blocking and well-performing services.

That being said, there have definitely been moments where Python showed some of its rough edges. To give an example of this, we had some trouble integrating a very synchronous library (i.e. blocking threads for long periods of time) into our framework of asynchronous code — it caused scheduler collapse that proved hard to avoid. It thereby starved e.g. HTTP and database operations, to the point where we decided to switch to a more complicated (but asynchronous) alternative because of it. A language with proper support for parallelism might have allowed for such interaction. In the case of Python, however, there is a clear divide in the ecosystem between asynchronous and synchronous libraries, and interaction between those can be tricky. This is something that will have to be kept in mind when implementing more advanced models as well, as Python's data science ecosystem is almost entirely synchronous. Long-running models might therefore have similar effects on the scheduler.

6.1.3.1. Software Improvement Group (SIG) Feedback

SIG provided us with the following feedback in response to the first deliverable:

> De code van het systeem scoort 4.3 sterren op ons onderhoudbaarheidsmodel, wat betekent dat de code boven marktgemiddeld onderhoudbaar is. We zien Unit Size vanwege de lagere deelscore als mogelijke verbeterpunt.
>
> Voor Unit Size wordt er gekeken naar het percentage code dat bovengemiddeld lang is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor dat elk onderdeel makkelijker te begrijpen, te testen en daardoor eenvoudiger te onderhouden wordt. Binnen de langere methodes in dit systeem, zoals bijvoorbeeld, zijn aparte stukken functionaliteit te vinden welke gerefactored kunnen worden naar aparte methodes.
>
> Jullie hebben niet zo veel lange methodes, maar bij `listen_on_topics()` in `__main__.py` is nog wat aanscherping mogelijk, bijvoorbeeld door het initialiseren van het object van het daadwerkelijke luisteren naar requests te scheiden.
>
> De aanwezigheid van testcode is in ieder geval veelbelovend. De hoeveelheid tests blijft nog wel wat achter bij de hoeveelheid productiecode, hopelijk lukt het nog om dat tijdens het vervolg van het project te laten stijgen.
>
> Over het algemeen is er dus nog wat verbetering mogelijk, hopelijk lukt het om dit tijdens de rest van de ontwikkelfase te realiseren.

In summary: maintainability of our code was scored at 4.3 out of 5 — above market average. Points of improvements were a single, overly lengthy function and a relatively low amount of tests compared to production code.

In response to the feedback from SIG, we:

- split up the offending long function into several smaller ones, and remained on the lookout for similar situations in any newly developed code
- added extra unit tests, including some for system components that had previously only been validated by manual and end-to-end tests

SIG evaluated these changes and countered with the following response:

> In de tweede upload zien we dat het project een stuk groter is geworden. De score voor onderhoudbaarheid is in vergelijking met de eerste upload licht gestegen.
>
> Bij Unit Size, het verbeterpunt uit de eerste upload, zien we een duidelijke en structurele verbetering. Daarnaast is het goed om te zien dat jullie niet alleen de bestaanden code hebben gerefactored, maar ook voor de nieuwe code andere normen hebben toegepast.

[2] Asynchronous HTTP Client/Server for asyncio and Python, see `https://aiohttp.readthedocs.io/en/stable/`

[3] Client for the Apache Kafka distributed stream processing system using asyncio, see `http://aiokafka.readthedocs.io/en/stable/`

[4] Database interface library designed specifically for PostgreSQL and Python/asyncio, see `https://magicstack.github.io/asyncpg/current/`

> Naast de toename in de hoeveelheid productiecode is het goed om te zien dat jullie ook nieuwe testcode hebben toegevoegd. De hoeveelheid tests ziet er dan ook nog steeds goed uit.
>
> Uit deze observaties kunnen we concluderen dat de aanbevelingen uit de feedback op de eerste upload zijn meegenomen tijdens het ontwikkeltraject.

In summary: we processed the previous feedback well, both in existing code (through refactoring) and in the significant amount of new code that was added. As a result, the maintainability score slightly increased (though no concrete number was given).

6.2. Process

6.2.1. Division of tasks

Design and development of our service's core architectural components was a shared team effort. This helped ensure that all components of the system align well, and that everyone has the required knowledge to improve any component. On top of that, each of us held responsibility for some smaller, more focused part of the full service. Communication with stakeholders in the company was largely split along these lines as well.

Data Science was the responsibility of Matthijs. This included communication with Daynan, the data scientist that collaborated with us. He is based in New York, which meant some slight inconvenience due to the differing time zones. However, because of the modular setup of the model there was a clear path towards model integration, allowing Daynan to focus on the model itself without bothering with too many software engineering details. Throughout the project, we built our own prototype model for testing purposes, whereas Daynan focused on exploring more complicated and future-minded ideas.

Gerben has invested the most time in the integration process with existing GeoPhy systems. The so-called Data Management Platform will be the environment our service will continue to live in after we leave, and we have had to invest a lot of time to align our service to work within this environment. To achieve this, Gerben had to contact several colleagues and gather information on the exact workings of the core architecture.

The graphical user interface used to display the results has been the primary responsibility of Robin. He devoted some time to setting up the simple overview, as well as connecting it to rest of the service. In addition, he has been the one maintaining most communication within the company related to the Perfect Comps service.

6.2.2. Planning

In the research report we included an initial project plan (see Appendix C for a copy). We have managed to keep this project plan as an outline for the course of our project. The division of tasks stated in the previous subsection has been a great help in this. From the start we have had different interests in this project, and this gave us the opportunity to work on the things we were most excited about. It helped us be able to work together as an effective team.

Sticking to the sprint planning as mentioned in the project plan has been a good choice. In chapter 5 it is clearly stated what was done in which sprint. To us it was clear what we expected to have completed by the end of each sprint. In hindsight we can say that we correctly estimated the amount of work needed to complete most tasks.

Development was planned in a loosely agile way — sprints involved little upfront planning, but we set and adjusted goals for ourselves several times per sprint, and kept a backlog of issues to pull from and achieve those.

6.2.3. Learning objectives

We want to briefly reflect on the learning goals of the course, which are as follows:

- Students can carry out an entire software development cycle with success, from researching solutions through testing the product, in a team of developers addressing a real-world problem.
- Students can effectively, in collaboration with a coach and a client, choose a development strategy and execute a development process according to that strategy.
- Students can establish the necessary quality requirements for a product and carry out the tests necessary to determine that the product fulfills those requirements.

- Students can present a complete and convincing explanation of the development process and the product results.

We feel that, over the course of this project we have definitely shown that we are able to carry out a full development cycle. As mentioned, this is one of the main learning objectives. This is closely connected to the collaboration with our coach and mainly client. Together with them we determined the actual approach to take within this project. GeoPhy has given us a great degree of freedom and has made clear that they are happy with the system's development process and the final result.

We have put a lot of effort into maintaining a high quality standard in our project. This was done through several code quality tools as well as a significant amount of testing code. By keeping ourselves to a high standard, we managed to received a high mark from SIG for our code quality.

The final learning objective is still open-ended, as we did not give our final presentation yet, so part of it is yet to be determined. However, we are happy with the results we have been able to show in this report. We believe that it is a good and convincing overview of our activities during the project.

6.3. Impact

Now that we have delivered our final solution to the problem we set out to solve, it will be interesting to see what its impact will be within GeoPhy and the real estate industry at large. The outcome of our project serves as a proof of concept for the idea of a Perfect Comparables service, and is not quite a customer-facing product yet. It does, however, show what the final product would look like, and the output it generates can already be used for internal purposes and customer evaluation.

The architecture of our service also has potential for productizing other kinds of models. Fundamentally, the process of gathering streamed data, feeding it to a pre-trained model, and storing that output for later recall is a very common requirement within the company, and can be applied to e.g. automated valuation models as well. A generalization of the Perfect Comps architecture could be a good fit for several other future GeoPhy services, especially when extended with model versioning (see section 8.2).

6.3.1. Ethical Implications

The Perfect Comps service aims to provide an objective truth. GeoPhy states that the traditional appraisers, who select sales comparables, are biased towards a certain locality as well as time frame. In our project description (Appendix A) it is stated that we need to develop a service that will convince their users that they shouldn't feel threatened by Machine Learning methods, but rather view them as another tool that industry experts can use to deliver value to customers. To do this, the Perfect Comps service needs to be a gateway product that mimics the behaviour of the appraiser, but gives clients new insights based on a more objective view on appraisals and sales comparables.

This means that we are, in essence, trying to replicate how an appraiser would approach reference comparables. The fact that these appraisers are biased by nature has given GeoPhy the intuition that it would be possible to replace them with a sophisticated algorithm. However, the Perfect Comps service is aiming to give a much broader view on sales comparables. It is geared towards a large data set — one that is not humanly creatable or understandable. Therefore, it is supposed to perform even better than traditional appraisers, and there is a clear motivation for its use beyond just automating away its human predecessors.

This gives us the sense that it is not ethically incorrect per se to develop the service. Its purpose is to help appraisals be based on facts and numbers instead of the gut feeling or experience in a certain market of an appraiser. The clients that traditionally use the help of appraisers will be given a better view on sales comparables. This, in turn, makes the sales comparables approach (which is currently very common) a lot more robust.

It is important to note that there will always be human interaction with the comparables that our algorithm will produce. The Perfect Comps product will serve as a tool to help the clients, and is merely a part of the complete appraisal process. It produces sensible explanations for each comparable that is selected, which means that we will be able to back our results, but also that clients are able to refuse the results based on this explanation.

7 Conclusion

Over the past quarter we have been involved in the design process and implementation of the Perfect Comps service for GeoPhy. We were able to make our own design choices during the development of this product. Through being given this responsibility we were able to put our engineering knowledge gained throughout the Bachelor into this system. In addition, we have seen what it entails to design and implement a product for a company instead of a teacher. We feel that we have shown the ability to come up with elegant solutions that the company did not expect, such as the choice to work with a serialized Python file to be used as a model. It showed our knowledge of the available techniques as well as a thorough understanding of the requirements set by the company. It enabled our data science colleague to work on his model rather easily.

In this chapter we want to answer the question that has been posed in chapter 1: *How can we design a service that accurately identifies sales comparables?* In chapter 5 we showed the process that we went through to complete this task. In chapter 6 we answered the subquestions proposed in chapter 1. It has been amazing to see that the architecture, as proposed in chapter 4, did not need much change when implementing this. It demonstrates that we are capable of designing a fully functional system and adhere to the design choices. It also means that the choices made in the first two weeks turned out to be realistic. If we look at the final product from a higher vantage point, we conclude that we have been able to fulfill the requirements, and therefore successfully completed the project. We have shown that we were able to transform requirements into a functional product that fulfills them.

Many industries lag behind the state of the art in how they benefit from the digital revolution of the past decades, and real estate is certainly one of them. Building technology for those in the long tail of the technology adoption lifecycle brings along unique challenges regarding trust generation, even though much of the low-hanging optimization fruit that has already been plucked in other sectors is still present. Certain aspects of the product development process, such as keeping change gradual and properly explaining how the more opaque parts of a product work, become immensely important. For the Perfect Comps service, this is especially the case, as it serves a purpose beyond its own capabilities: to help push the adoption of its more advanced peer products within GeoPhy, and eventually render itself (partially) obsolete. Additionally, the company has decided upon a streaming data architecture that is highly dynamic in nature — something that, while very useful, is not well-matched to many traditional modes of machine learning. It needs to be taken into account continuously, both during design and implementation. The combination of all this makes for a very intriguing challenge, and solving part of it during our project was an interesting and enjoyable experience. With our final product, GeoPhy will be able to deliver further innovation to their clients, and advance the state of the art in their industry — something we are glad to have contributed to.

8 Recommendations

Now that we have concluded our project, it is time to look forward. In this final chapter we would like to give a set of recommendations to GeoPhy that it could use to improve the product that we have delivered. The chapter is sectioned based on the areas where improvements are possible based on our view of the project.

8.1. Scalability

On the point of of scalability, there are still some parts where our system can be improved. The statistics store might become a bottleneck, as each row in its database table is written to for every entity that is processed. Running multiple instances of the database is feasible, but would need custom conflict resolution when two instances diverge. That could, of course, be handled in a straightforward way, by recalculating the statistics whenever they diverge. However, that becomes problematic at large data set sizes. Alternatively, it is possible to switch to keeping track of statistics only for a sampled fraction of input data. On large enough data sizes, this should not affect the distribution of the data set meaningfully, and the sample fraction can be lowered to meet computational demands. Finally, statistics updates could be stored in a log-like structure instead. The logs can be compacted frequently to limit their size, and the current state statistics store can be reconstructed from those logs in case of divergence.

Another possible bottleneck is the indexer, which can become overloaded when ingesting large amounts of data. This can impact the availability of the comparator. One way to fix this would be to use a Kafka stream as a backpressure mechanism for database writes, with workers that pull from the stream at a rate that can be handled by the indexer. This would leave the preprocessor free to run on as many instances as is desired. If this still is too resource-intensive or unstable, index search operations could make use of read replicas of the index database instead, which would dedicate the writable database instance to index insertions. Heavy writing would not meaningfully impact the performance of the comparator anymore at that point, and additional latency is negligible. Finally, a different type of database could be used that can be scaled out better, although it is not immediately obvious what database that would need to be.

In the current solution, all information is stored for further reference, because it might at some point become relevant. This means that the central data store is replicated inside our service. This is a sub-optimal situation, because there are more efficient ways to replicate RDF databases. Each service within GeoPhy that is similar to our service faces this problem. Having a separate store that contains all known data for each service is very inefficient. An option here is to create a separate service which is responsible for keeping track of sets of data that are considered complete by other services.

8.2. Model improvements

A more obvious improvement in future iterations is the model. This is mostly the responsibility of Daynan, but a significant amount of preliminary research for that has been done by us. Daynan has read our findings in the research report and is looking to implement, among other things, a relational auto-encoder soon. Better support for sparse feature vectors is also being explored, through e.g. advanced data imputation techniques. Sparse data occurs frequently, but should not raise any errors when processing (which eliminates the incomplete property from the search space). For now, this is still the case, as the system requires certain important features to be present unconditionally in order to build the feature vector.

Another point of improvement might be the logic used in the indexer. The k-NN used in the current system is sufficient, but better alternatives might exist that have not yet been evaluated well. It could be interesting to investigate the use of different methods of clustering comparables that might take into account more complicated patterns in the data set. k-NN in its current incarnation is also not that tunable itself — whatever the preprocessor generates cannot be changed dynamically (to weigh features differently in response to user input, for example) because the index structure is pre-calculated on insertion. Eliminating this constraint might help offer comparables that are better tuned to each individual customer's demands.

Finally, model versioning could be a valuable addition to the service, so that updated models don't cause untraceable changes in the service's behavior. Any model that would be exposed to customers at the initial release of Perfect Comps is likely to change over time, both in response to user feedback and due to possible new modeling techniques or additional data. This will result in service output shifting over time, which may be a point of confusion for customers. Some of them might even require model consistency as part of agreements, or want to compare model output across versions to see if we still meet their demands. It might also be useful to maintain different models for different categories of buildings (think housing versus office space) that are not inherently comparable, or to develop different models that focus on other aspects of comparability (financial versus location, for example). Finally, once a group of customers starts relying on the product, the versioning system can be used for A/B-testing multiple models in parallel in order to measure the demands of customers more objectively. All of these use cases can be captured by storing/maintaining multiple models in parallel, and building the interface functionality required to switch between them.

8.3. Explanation

The silos that are present in the current user interface are mainly used as a proof of concept. They incorporate some well-designed statistics, but this has not been tested with users. The comparability metrics should be tuned if we want to present these figures to an actual client. In that case, GeoPhy could decide to change the numbers and tweak them to their liking.

We decided to explain model output without taking into account the model itself. This is necessary because machine learning technologies often prohibit meaningful explanation of their process, and ignoring those solutions felt overly restrictive. However, ML explainability is a topic of heavy research right now, and future developments in the area might make it feasible to merge model and explanation into a single entity to better reflect how comparables are actually generated. Alternatively, it might make sense to move the explanation logic from hand-tuned to more automated, using the ontological structure of input data that GeoPhy has already extensively incorporated.

8.4. Productization

To arrive at a production-ready final product/model and achieve product-market fit, it will be necessary to put Perfect Comps to the test, and get some user feedback (whether provided by them or measured in some more objective way). Initially, generating different sets of comparables and having users score or rank those could help steer model training and feature weights towards user priorities. Once the service is in production, inclusion of a feedback loop could help keep model quality high, and allow learning from service users how to improve our model or prioritize/structure new products.

The most interesting idea might be to see if it is possible to set up a highly dynamic feedback loop on input data by directly exposing the Perfect Comps service to its end users over the internet. This would require eliminating all human intervention on the GeoPhy side, and making the output of the service queryable directly in the form of a more search engine-like product. Whether this is desirable from a pure business perspective is another question entirely, and volume/quality of data is highly dependent on user count and demographics. If such open questions are answered positively, however, modifying the service to facilitate it would open up opportunities for collecting more objective data. This can be used to train more advanced or even personalized models. There is also research out there on models that have a human feedback loop even during their execution (as opposed to in training data only). This is still rather cutting-edge, but especially for customers that do not trust algorithms all that much, having the feeling that they can influence the model's process in real time could make all the difference. This could help push the product through the early adopter barrier and towards more general acceptance. To mention a few interesting papers in the field: some of the possible modes of interaction were described by Tintarev and Masthoff (2007), while Brown et al. (2012) consider interactively learning a distance function specifically.

A Project Description

A.1. Introduction

GeoPhy is one of the leading companies when it comes to Automated Valuation for commercial real estate globally. We collect data at a large scale, integrate this using our DataManagementPlatform and feed this to our Machine Learning (ML) models for providing valuations of commercial building around the globe. We see technology and novel technical solutions as our key differentiator and as our main driver to be able to do this at a global scale, providing a reliable and accurate valuation of all properties in our system at a daily interval.

A.2. Assignment

For traditional, manual real estate appraisals the appraiser is required to provide a number of comparable properties (the 'Comps'). These comps act as a benchmark for the valuation as well as a provider of context in the final appraisal report. The ultimate goal for GeoPhy is to replace these reports by an automated valuation. Although technically we are capable of providing this value we see that the market in general is not yet ready for this level of automation (with the exception of some frontrunners). In current practice the comps are selected manually by the appraiser and are typically chosen within a ten mile radius of the subject property (assuming USA as the market). This manual selection is biased by the knowledge of the market by the appraiser and the amount of transactions that happened in that area. Since these comps are of great importance to the final valuation given and the overall impression provided we argue that providing an objective approach to the selection would benefit the overall quality of the report and would provide a low threshold introduction into ML driven valuations. For this assignment we will focus on US data in the MultiFamily space (apartment buildings) The selection of relevant comps should not be limited to the ten mile radius currently used. We argue that a perfect comparable could be on the other side of the country or even in a other moment in time (market conditions five years ago in a certain city could be identical to the current situation of my subject property).

A.3. Challenges

To provide a restful API that, given a building address, is able to provide the top 10 'best' comps from our building database. For the selection you need to take into account the building characteristics, the financial metrics, the contextual characteristics and time as possible features for selection. Our initial point of reference for the solution are recommendation engines with as key challenges:

- The large amount of features (est. 500-1000) including time
- The fact that we use a streaming data platform: data is always in motion and changing, how do you keep your model up to date
- Response times should be < 1sec
- Explore the various approaches to recommendations engines, chose and implement the best fit

A.4. Technologies

- The service to be developed will sit in our DataManagementPlatform
- Consume all relevant data from Apache Kafka topics
- Have its own data store (to be determined, internally we use a wide range of different datastores)
- Expose a RESTful API with a base frontend to act as a proof of concept
- Deploy as containerised service on our cloud infrastructure (AWS)
- The company has a focus on using Scala and Python for all services but this is not a hard requirement

Robin van Heukelum - robin.van.heukelum@gmail.com

Interests: *Business Communication, Product Ownership, Client Interaction*
Contributions: *Front-end development, Client communication*

Gerben Oolbekkink - g.j.w.oolbekkink@gmail.com

Interests*: Complexity Theory, Software Architecture, Concepts of Programming Languages*
Contributions*: Back-end development, System integration*

Matthijs Wolting - matthijswolting@gmail.com

Interests: *Software Architecture, Distributed Systems, Embedded Systems, Artificial Intelligence*
Contributions: *Back-end development, Data science*

Context

Comparable selection does not have an objective ground truth, which complicates or even prohibits the use of many machine learning algorithms that could otherwise have been used. Additionally, the outcome of the service needs to be explainable to its users — it cannot be completely opaque. Finally, our service needed to integrate with a streaming data platform, with incremental new data that arrives continuously and needs to be incorporated into the service's model and output.

For traditional, manual real estate appraisals, the appraiser is required to provide a number of comparable properties (the 'Comps'). These comps act as a benchmark for the valuation as well as a provider of context in the final appraisal report. Traditionally, these comps are selected manually by an appraiser based on recent transactions within a ten mile radius. This manual selection is biased by the appraiser's market knowledge and the amount of transactions in the area. To replace this process, we developed an automated comparable selection service that does so based on objective characteristics, without restricting itself to a small spatial and/or temporal slice of the market.

Research

Our research phase focused on three aspects: determining the possible algorithms for selecting comparable properties given the constraints of explainability and a streaming environment, how to explain the output of the chosen algorithm to the user, and how to build a service around the chosen model that consumes a stream of input data and can generate a set of comparable properties ad-hoc.

Perfect Comps Service

Process

Our process was based on agile methodology, with two-week sprints in which we gradually expanded our service into a fully functional proof of concept. Challenges were encountered while developing the logic to incrementally construct a database of real estate properties from the stream of data. These were resolved by switching from a document store to an RDF database, which better matched the flow of data coming in.

Future

As a future improvement, the current model used by the service is fairly simple and can most likely be improved upon once more data is available. Additionally, due to the lack of a ground truth it will be important to tune both comparable selection and explanation metrics in response to user testing.

Product

The final product consists of several microservices, each of which handles part of the problem domain and can be scaled out independently. A REST API and a web front-end are accessible to its users. The system was tested using both unit tests and end-to-end testing, whereas the model was refined by scoring output on closeness of features indicative of similarity.

Date of presentation
July 3rd, 2018

The final report for this project can be found at: http://repository.tudelft.nl/

GEOPHY

Sander Mulders

CTO
s.mulders@geophy.com

Cynthia Liem

Intelligent Systems
Multimedia Computing Group
c.c.s.liem@tudelft.nl

TU Delft

C Project Plan

C.1. Approach

To start off, we will familiarize ourselves with the task, as well as possible problems and solutions as found in scientific literature. By looking into literature relevant to comparable selection and/or similar problem spaces, we will map the possible solutions and be able to select a promising combination of them for our purposes.

This will be followed by a phase of prototyping and experimenting, with the goal of developing a functional comparable selection model. This phase will be one of rapid iteration, where we attempt to validate possible solutions as quickly as possible and expand on them if initial results are good. Focusing on the model first is important, as this is likely to be the area with the most technology risk — success or failure of our product hinges on that of the model we develop.

Finally, we will integrate our model with the infrastructure used by GeoPhy, so that it can be used and maintained as a product. While we might previously have experimented mostly using on-disk files or ad-hoc databases, GeoPhy has a standardized system for data ingestion, processing, etc. that we need to connect to, so that our model is continuously provided with the latest data sets and can remain relevant. The final solution should be able to remain up to date automatically.

C.2. High-level planning

Sprint	**Activities**
0 (week 1 - 2)	• Project Plan • Literature Review • Research Report
1 (week 3 - 4)	• Explore the available data • Experiment with naive k-NN • Experiment with dimensionality reduction • Experiment with importance extraction from existing comps • Build prototype
2 (week 5 - 6)	• Select and polish a model for integration • Prepare code for first SIG check
3 (week 7 - 8)	• Convert model to ingest data from existing GeoPhy infrastructure
4 (week 9 - 10)	• Prepare code for second SIG check • Finalize report

C.3. Deadlines

May 7th, Monday	Research Report
June 1st, Friday	SIG Code upload 1
June 22nd, Friday	SIG Code upload 2
June 25th, Monday	Final Report

Bibliography

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). The r*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, volume 19, pages 322–331. Acm.

Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517.

Brown, E. T., Liu, J., Brodley, C. E., and Chang, R. (2012). Dis-function: Learning distance functions interactively. In *2012 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 83–92.

Buja, A., Swayne, D. F., Littman, M. L., Dean, N., Hofmann, H., and Chen, L. (2008). Data visualization with multidimensional scaling. *Journal of Computational and Graphical Statistics*, 17(2):444–472.

Castelli, V., Thomasian, A., and Li, C.-S. (2003). Csvd: Clustering and singular value decomposition for approximate similarity search in high-dimensional spaces. *IEEE Transactions on knowledge and data engineering*, 15(3):671–685.

Chu, W. and Cai, D. (2017). Stacked similarity-aware autoencoders. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pages 1561–1567. AAAI Press.

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB conference, Athens, Greece*, pages 426–435. Citeseer.

Covington, P., Adams, J., and Sargin, E. (2016). Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, New York, NY, USA.

Detweiler, J. H. and Radigan, R. E. (1996). Computer-assisted real estate appraisal: A tool for the practicing appraiser. *The Appraisal Journal*, 64(1):91.

Diaz III, J. (1990). The process of selecting comparable sales. *The Appraisal Journal*, 58(4):533.

Downie, M.-L. and Robson, G. (2008). Automated valuation models: an international perspective. *RICS Automated Valuation Models Conference: AVMs Today and Tomorrow.*

Guttman, A. (1984). *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM.

Herlocker, J. L., Konstan, J. A., and Riedl, J. (2000). Explaining collaborative filtering recommendations. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 241–250. ACM.

Indyk, P., Motwani, R., Raghavan, P., and Vempala, S. (1997). Locality-preserving hashing in multidimensional spaces. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 618–625. ACM.

Jo, J., Seo, J., and Fekete, J.-D. (2017). A progressive kd tree for approximate k-nearest neighbors. In *Workshop on Data Systems for Interactive Analysis (DSIA).*

Kibriya, A. M. and Frank, E. (2007). An empirical comparison of exact nearest neighbour algorithms. In *European Conference on Principles of Data Mining and Knowledge Discovery*, pages 140–151. Springer.

Klema, V. and Laub, A. (1980). The singular value decomposition: Its computation and some applications. *IEEE Transactions on automatic control*, 25(2):164–176.

Klyne, G. and Carroll, J. J. (2006). Resource description framework (rdf): Concepts and abstract syntax. *W3C Recommendations.*

Kok, N., Koponen, E.-L., and Martínez-Barbosa, C. A. (2017). Big data in real estate? from manual appraisal to automated valuation. *The Journal of Portfolio Management*, 43(6):202–211.

Kreps, J., Narkhede, N., Rao, J., et al. (2011). Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7.

Liu, T., Moore, A. W., Yang, K., and Gray, A. G. (2005). An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, pages 825–832.

Makhzani, A. and Frey, B. (2013). K-sparse autoencoders. *arXiv preprint arXiv:1312.5663*.

McKerns, M. M., Strand, L., Sullivan, T., Fang, A., and Aivazis, M. A. G. (2012). Building a framework for predictive science. *CoRR*, abs/1202.1056.

Muja, M. and Lowe, D. G. (2014). Scalable nearest neighbor algorithms for high dimensional data. *IEEE transactions on pattern analysis and machine intelligence*, 36(11):2227–2240.

Naidan, B., Boytsov, L., and Nyberg, E. (2015). Permutation search methods are efficient, yet faster search is possible. *Proceedings of the VLDB Endowment*, 8(12):1618–1629.

Omohundro, S. M. (1989). *Five balltree construction algorithms*. International Computer Science Institute Berkeley.

Pagourtzi, E., Assimakopoulos, V., Hatzichristos, T., and French, N. (2003). Real estate appraisal: a review of valuation methods. *Journal of Property Investment & Finance*, 21(4):383–401.

Pearson, K. (1901). On lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572.

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144. ACM.

Rifai, S., Vincent, P., Muller, X., Glorot, X., and Bengio, Y. (2011). Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 833–840. Omnipress.

Roweis, S. T. and Saul, L. K. (2000). Nonlinear dimensionality reduction by locally linear embedding. *Science*, 290(5500):2323–2326.

Schölkopf, B., Smola, A., and Müller, K.-R. (1998). Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319.

Sellis, T., Roussopoulos, N., and Faloutsos, C. (1987). The r+-tree: A dynamic index for multi-dimensional objects. Technical report, University of Maryland.

Shih-MingYou and Chang, C.-o. (2009). Weight regression model from the sales comparison approach. *Property Management*, 27(5):302–318.

Smith, A. and Nolan, J. J. (2018). The problem of explanations without user feedback. *CEUR Workshop Proceedings*.

Son, J. and Kim, S. B. (2017). Content-based filtering for recommendation systems using multiattribute networks. *Expert Systems with Applications*, 89:404–412.

Sun, Y., Han, J., Yan, X., Yu, P. S., and Wu, T. (2011). Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment*, 4(11):992–1003.

Tang, B., Yiu, M. L., and Hua, K. A. (2016). Exploit every bit: Effective caching for high-dimensional nearest neighbor search. *IEEE Transactions on Knowledge and Data Engineering*, 28(5):1175–1188.

Tenenbaum, J. B., De Silva, V., and Langford, J. C. (2000). A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323.

Tintarev, N. and Masthoff, J. (2007). A survey of explanations in recommender systems. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 801–810.

Todora, J. and Whiterell, D. (2002). Automating the sales comparison approach. *Assessment Journal*, 9(1):25–25.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*.

Van Dell, K. D. (1991). Optimal comparable selection and weighting in real property valuation. *Real Estate Economics*, 19(2):213–239.

Vincent, P., Larochelle, H., Bengio, Y., and Manzagol, P.-A. (2008). Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM.

Wang, W., Huang, Y., Wang, Y., and Wang, L. (2014). Generalized autoencoder: A neural network framework for dimensionality reduction. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 490–497.

Wang, Y., Yao, H., and Zhao, S. (2016). Auto-encoder based dimensionality reduction. *Neurocomputing*, 184:232–242.

Yang, C., Yu, X., and Liu, Y. (2014). Continuous knn join processing for real-time recommendation. In *Data Mining (ICDM), 2014 IEEE International Conference on*, pages 640–649. IEEE.

Yue, H. H. and Tomoyasu, M. (2004). Weighted principal component analysis and its applications to improve fdc performance. In *Decision and Control, 2004. CDC. 43rd IEEE Conference on*, volume 4, pages 4262–4267. IEEE.