PyExplore: Query Recommendations for Data Exploration without Query Logs

Apostolos Glenis
Athena RC
Athens, Greece
aglenis@athenarc.gr

Georgia Koutrika
Athena RC
Athens, Greece
georgia@athenarc.gr

ABSTRACT

Helping users explore data becomes increasingly more important as databases get larger and more complex. In this demo, we present PyExplore, a data exploration tool aimed at helping end users formulate queries over new datasets. PyExplore takes as input an initial query from the user along with some parameters and provides interesting queries by leveraging data correlations and diversity.

CCS CONCEPTS

• **Information systems → Data mining**.

KEYWORDS

data exploration,query recommendations,clustering

ACM Reference Format:
Apostolos Glenis and Georgia Koutrika. 2021. PyExplore: Query Recommendations for Data Exploration without Query Logs. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 18–27, 2021, Virtual Event, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3448016.3452762

1 INTRODUCTION

Data consumers such as analysts, scientists (biologists, astrophysicists etc) and data scientists, access data sets of interest, looking for answers, insights, patterns. Given the volume and complexity of data as well as the fact that SQL is cumbersome for data exploration, users often struggle with finding what queries to ask when they interact with a data set. *Query recommendations* come to the rescue [13] as they help the user focus their search (e.g., by recommending possible completions for a query [9, 19]) or offer an alternative search path (e.g., recommending similar queries based on user query behavior [4]). To do so, existing work is either based on data analysis [5] or query log analysis [4, 9, 19]. *Query logs* have been used either to mine frequent patterns and query dependencies to find completions for user queries [9, 19] or to apply typical recommendation strategies such as matrix factorization and latent models [4] to find queries that could be related to what the user is currently looking for. In the absence of query logs, approaches are based on *data analysis* [5]. The idea of splitting the initial user query in meaningful areas has been introduced in [11, 17].

In our work, we focus on *query completion recommendations* in a *cold-start setting*, i.e., without relying on query logs, an inherently challenging problem. We present a new query recommendation tool, PyExplore, that takes as input a SQL query and returns a set of ranked queries, each of them focusing on a subset of the initial query results that can help the user navigate the data space. PyExplore is based on the observation that people better perceive fewer dimensions, typically 2-3, and *understand patterns in low-dimensional spaces*. PyExplore first finds 'interesting' subsets of query attributes, i.e., that they exhibit *correlation* or *diversity*. Then, for each such subset, it clusters the data based on the values of the respective attributes. Our approach is based on the idea of guiding the users with the help of cluster analysis introduced in [15] and applied in the context of query recommendations in [11, 17]. Each cluster is mapped to a SQL query, which becomes a candidate recommendation to be shown to the user. Essentially, each candidate query is generated by augmenting the WHERE clause of the initial query with new clauses that describe those cluster boundaries with the help of a decision tree classifier.

We are working with three collaborators representing three user communities, namely astrophysics, biology and policy making, on how query recommendations can enhance their data exploration tasks. To illustrate how PyExplroe works, we provide an example of a real use case with astrophysicists.

Motivating Example. Sri, an astrophysicist, explores astronomical objects in SDSS[1], a large sky survey database. A single table in SDSS, $photo\_obj$, contains several millions of rows and several attributes. Sri would like to examine a part of the sky so she starts with a query that defines the area she is interested in. First, PyExplore finds how the attributes in the query results may correlate. For example, it finds that *relative ratios and strength of emission lines* are correlated; also, objects are correlated based on *luminosity and surface temperature*. Then, PyExplore clusters the query results for each set of correlated attributes. Clustering aims at producing meaningful subsets of the query results. To generate query recommendations, these clusters (per set of correlated attributes) are fed into a decision tree classifier. The conditions in each WHERE clause describe those cluster boundaries according to the split points of the decision tree classifier. In our example, PyExplore would generate query recommendations for *similar objects in terms of their relative ratios and strength of emission lines* as well as for *similar objects in terms of luminosity and surface temperature*. Sri chooses a particular recommendation to see similar objects based on the former group

[1]https://www.sdss.org/

of attributes. Then, PyExplore will generate new query recommendations for this new query. Finally, Sri may discover that green pea emission line ratios are similar to high redshift galaxies. This is a typical example of how PyExplore-type query recommendations can help data exploration and discovery. □

Challenges. Several challenges arise, including: (*a*) finding interesting subsets of attributes, (*b*) handling different types of attributes such as numerical, categorical, string, (*c*) scoring query recommendations, and (*d*) scaling to large datasets.

Contributions. PyExplore makes the following novel contributions with respect to previous related work [11, 17]. To find interesting subsets of attributes, it leverages two different intuitive concepts: data *correlation* and *diversity*. To handle data set heterogeneity, it *handles mixed numeric and categorical attributes*. Furthermore, to handle textual data, PyExplore extends the query recommendation framework to a *workflow for textual attributes*. It vectorizes the textual attributes and provides recommendations based on their numeric representation. PyExplore can determine whether a column is categorical or string by looking at the number of unique values. The user can then choose whether to use the mixed data or the string workflow. To rank recommendations, it *leverages the clustering quality metrics*. Finally, since clustering is a time-consuming process, PyExplore uses *three ways of improving execution time*: sampling on rows and columns of the data set, precomputing the vectorization model and associations for the textual attributes and an approximate workflow.

Demonstration. PyExplore also features a user-friendly UI that allows the user to explore the query recommendations and a data set. The tool allows the user to *progressively build different exploration paths* by asking and following query recommendations. The user can inspect results and useful visualizations as well as backtrack an exploration path to follow a different one. During the demonstration, the participants will have the chance to dive into *different exploration scenarios in several data sets* as well as look into the algorithms and *evaluate different recommendation configurations*. We will demonstrate PyExplore with real-world data exploration use-cases such as astrophysics and policy making.

2 QUERY RECOMMENDATION MODEL

We consider an SPJ query Q over a database D of the form:

SELECT **A** FROM **T** WHERE **P**

where **T** is a set of tables joined for the query, **A** is a subset of the table attributes projected in the query result, and **P** is a conjunction of selection predicates.

Given Q, the objective of query recommendation is to generate a *ranked set of queries* $\mathbf{Q} = \{Q_i | i = 1 \ldots k\}$, where Q_i has the form:

SELECT **A** FROM **T** WHERE $\mathbf{P_i}$

where $\mathbf{P_i}$ is a conjunction of predicates over **T** that is a superset of **P**. In other words, in each Q_i, the WHERE clause is augmented with new clauses. If the initial query did not contain a WHERE clause, a new WHERE clause is built.

3 SYSTEM OVERVIEW

3.1 Interesting Subsets of Attributes

The first step is to find 'interesting' subsets of query attributes. PyExplore leverages two notions: attribute correlation and diversity.

Correlation-based. Correlation is the measure of how two features are correlated. For example, the month-of-the-year is correlated with the average daily temperature, and the hour-of-the-day is correlated with the amount of light outdoors. Data scientists are interested in correlated attributes to highlight relationships between attributes of the data set.

First, PyExplore computes the correlation of each pair of attributes in the query results. One challenge is how to deal with different types of attributes. For comparison between numerical attributes, it uses *Pearson correlation*[2], for categorical-categorical, it uses *Cramer's V*[2], and for categorical-numerical, it uses *Correlation Ratio*[3]. To make all correlation metrics in the same range, i.e., [0, 1], we take the absolute value of the Pearson Correlation.

Then, the inverse of the absolute value of the correlation matrix is used as a distance matrix, which is given as input to a clustering algorithm that creates clusters of correlated attributes. PyExplore uses two options for clustering correlated attributes: (a) *hierarchical clustering with complete linkage*[4] takes as input the maximum number *size_max* of attributes per cluster and decides the number of clusters accordingly, and (b) OPTICS [1], which is a density-based algorithm that decides how many clusters to create and also clusters all outliers together. This big cluster with outliers is ignored by the recommendation algorithm.

Diversity-based. Intuitively, an attribute that has a diverse set of values is interesting because it allows the user to explore a larger part of the initial query results compared to a less diverse attribute. To compute diversity for numerical columns, PyExplore uses the *normalized Shannon entropy*[5][18]. For categorical columns, it computes the ratio between the unique values in the column and the total rows in the column. Then, subsets of diverse attributes up to a *size_max* size are generated in a greedy manner.

Note that both correlation and diversity are computed on-the-fly on the results of the initial user query.

3.2 Generating Query Recommendations

Result Clustering. For each subset of attributes identified by the first step, PyExplore clusters the initial query results using the values of the attributes in the subset. It uses two options. The first is K-means with scaling and encoding categorical values as dummy variables, as proposed in [17]. However, encoding categorical values as dummy variables can lead to increased time and space complexity for data sets with high-cardinality categorical values. To overcome this problem, PyExplore uses K-modes [7]. Specifically, to enable the clustering of categorical data in a fashion similar to k-means, the algorithm proposed in [6] uses a simple matching dissimilarity measure, replaces the means of clusters with modes, and uses a frequency-based method to update modes in the clustering process to minimise the clustering cost function. The algorithm proposed in [8], through the definition of a combined dissimilarity measure, further integrates the k-means and the algorithm presented in [6] to allow for clustering objects described by mixed numeric and categorical attributes. Note that algorithms such as OPTICS are very slow for this phase, and hence were not selected. The parameter

[2]https://en.wikipedia.org/wiki/Pearson_correlation_coefficient
[3]https://en.wikipedia.org/wiki/Correlation_ratio
[4]https://en.wikipedia.org/wiki/Hierarchical_clustering
[5]https://en.wikipedia.org/wiki/Entropy_(information_theory)

k, i.e., the number of clusters, can be specified by the user, while we investigate existing solutions for automatically determining k. One important challenge here is scaling to large data sets.

Query Generation. For each subset, the resulting cluster labels are fed into a decision tree classifier to produce the split points of the data. The resulting split points are used to create the recommended SQL queries. More concretely, PyExplore traverses the decision tree from the leaves up to the root, and for each path from the starting leaf to the root, it generates an output query. The conditions of the WHERE clause of each query describe the cluster boundaries as they are described by each path in the decision tree.

Since PyExplore uses clustering to obtain partitions of the data space, it leverages clustering quality metrics to obtain a ranking of the produced recommended queries. Specifically, it uses density as a quality metric for clustering, defined as follows:

$score = \frac{betweenss}{totss}$

where $betweenss$ is the between-cluster sum of squares and $totss$ is the total sum of squares. $betweenss$ can also be written as $betweenss = totss - withinss$ where $withinss$ is the total within-cluster sum of squares. Higher density score is better, meaning that the respective query describes a very dense area of the data.

3.3 Handling Textual Data

For *finding interesting subsets* of textual attributes, PyExplore *treats each query attribute as a document*. Then, these are mapped to numeric vectors using one of the following options:

- *CountVectorizer* [14] creates a histogram for each document.
- *TF-IDF* [14] essentially scales the histogram of the document by the number of documents in the corpus that contain the word, which aims to weigh down words that appear frequently in all the documents.
- *doc2vec* [10] utilizes word2vec [12] to create numerical representations of documents while still retaining their semantic properties. By using doc2vec, the similarity metric encapsulates the meaning of the words.

In our tests, doc2vec provided the best results because it determines relations between words of the attributes and not just frequencies.

Next, for the *query generation*, *each row in the initial query results is treated as a separate document.* For the clustering, any of the three above-mentioned vectorization algorithms can be used to transform each row to a vector representation. K-means is used for clustering.

To make the features of the decision tree interpretable, we use a CountVectorizer to vectorize the dataset for the decision tree. CountVectorizer essentially returns a histogram for each row. Using the vocabulary and the features of the splits, PyExplore creates the query recommendations based on the textual data. More specifically, we take the split points from the rules of the decision tree and decide whether we add the feature into a LIKE clause or a NOT LIKE clause. Features that appear as splits with histogram values less than 1 are added in a NOT LIKE clause; splits with histogram values equal or greater than 1 are added in a LIKE clause. *However*, since we concatenate each row to produce a single document, we do not know anymore to which attribute the feature corresponds. To address that, *a post-processing step* looks into the set of words of each attribute and determines to which attribute each feature corresponds *with the help of indexing*.

Another problem that might arise is for data sets with large vocabulary size. Both the CountVectorizer and TF-IDF produce sparse matrices. Both the K-Means implementation and the Decision tree classifier can handle sparse datasets so memory consumption should not be a problem even for large vocabulary sizes. doc2vec produces features of bounded size so vocabulary size is not a problem. So if the vocabulary size is large the user could use doc2vec for subset selection and clustering. Only the last part of the algorithm that requires CountVectorizer could present a problem in terms of execution time for large vocabulary sizes.

3.4 Dealing with Large Data Sets

PyExplore uses sampling, precomputing the vectorization model, and an approximate workflow to deal with large datasets.

- The user can use sampling on rows and columns of the dataset to speed up execution of PyExplore.
- Precomputing the doc2vec model can significantly improve the execution time of workflows using it.
- There is an approximate version of the PyExplore workflow, where the clustering step is performed by Mini-Batch K-means [16] and the decision tree classifier uses Hoeffding Trees [3].

4 PYEXPLORE UI

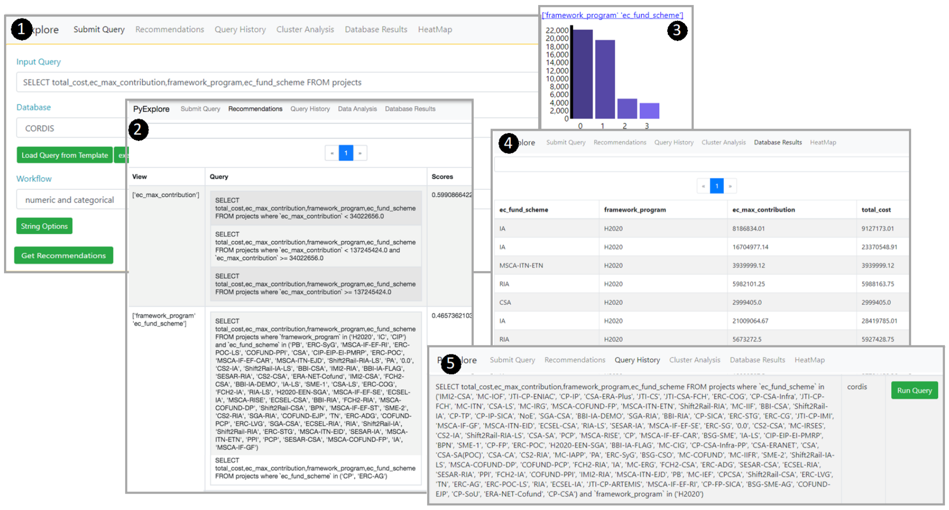
The PyExplore UI offers the user an interactive way to explore data sets. Figure 1 shows example screenshots. The user can start with a data set and a query ①. PyExplore returns query recommendations along with their score ②. The user can select to see how the query results are clustered for each subset of attributes selected by the algorithm ③. The user can execute a recommended query. The tool shows the first N (default value is 10) rows of the results of the query ④. The user can use a recommended query as a basis to ask for new recommendations and go deeper in the data. At any point, the user can visit their history of queries and pick a query to follow a different exploration path ⑤. The tool allows the user to modify the output of the recommendation algorithm by specifying the number of attributes, the number of recommendations, as well as try different algorithms and different types of recommendations based on mixed numerical and categorical or textual data.

5 DEMONSTRATION

Datasets. We have integrated several datasets, which will be used during the demonstration: (a) The movies dataset from Kaggle[6] provides information about the gross earnings of movies; (b) The IBM Car sales dataset[7] contains data about car sales; (c) The Intel dataset [8] contains information from sensors in the Intel Berkeley lab; (d) The CORDIS dataset[9] contains projects and related organisations funded by the European Union under the Horizon 2020 framework programme; (e) the SDSS dataset [10] contains astrophysics data.

Each data set offers different opportunities and challenges for recommendations. In our demonstration, we will use these data sets

[6]https://www.kaggle.com/rounakbanik/the-movies-dataset/version/7#movies_metadata.csv
[7]https://www.kaggle.com/thatbrock/ibm-watson-saleswinloss
[8]http://db.csail.mit.edu/labdata/labdata.html
[9]https://data.europa.eu/euodp/en/data/dataset/cordisH2020projects
[10]http://skyserver.sdss.org/dr16/en/home.aspx



SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects

View	Query	Scores

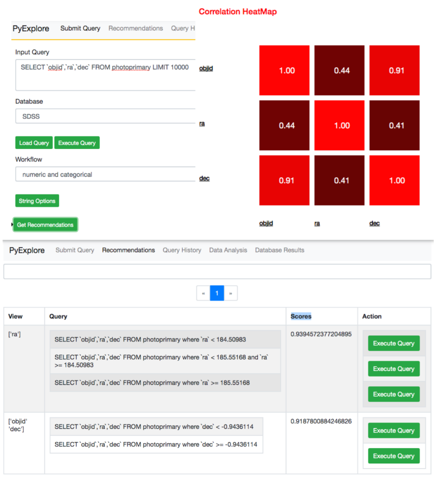
['ec_max_contribution']	SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects where \`ec_max_contribution\` < 34022656.0
	SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects where \`ec_max_contribution\` < 137245424.0 and \`ec_max_contribution\` >= 34022656.0
	SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects where \`ec_max_contribution\` >= 137245424.0
		0.5990866422

['framework_program' 'ec_fund_scheme']	SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects where \`framework_program\` in ('H2020', 'IC', 'CIP') and \`ec_fund_scheme\` in ('PB', 'ERC-SyG', 'MSCA-IF-EF-RI', 'ERC-POC-LS', 'COFUND-PPI', 'CSA', 'CIP-EIP-EI-PMRP', 'ERC-POC', 'MSCA-IF-EF-CAR', 'MSCA-ITN-EJD', 'Shift2Rail-RIA-LS', 'PA', '0.0', 'CS2-IA', 'Shift2Rail-IA-LS', 'BBI-CSA', 'IMI2-RIA', 'BBI-IA-FLAG', 'SESAR-RIA', 'CS2-CSA', 'ERA-NET-Cofund', 'IMI2-CSA', 'FCH2-CSA', 'BBI-IA-DEMO', 'IA-LS', 'SME-1', 'CSA-LS', 'ERC-COG', 'FCH2-IA', 'RIA-LS', 'H2020-EEN-SGA', 'MSCA-IF-EF-SE', 'ECSEL-IA', 'MSCA-RISE', 'ECSEL-CSA', 'BBI-RIA', 'FCH2-RIA', 'MSCA-COFUND-DP', 'Shift2Rail-CSA', 'BPN', 'MSCA-IF-EF-ST', 'SME-2', 'CS2-RIA', 'SGA-RIA', 'COFUND-EJP', 'TN', 'ERC-ADG', 'COFUND-PCP', 'ERC-LVG', 'SGA-CSA', 'ECSEL-RIA', 'RIA', 'Shift2Rail-IA', 'Shift2Rail-RIA', 'ERC-STG', 'MSCA-ITN-EID', 'SESAR-IA', 'MSCA-ITN-ETN', 'PPI', 'PCP', 'SESAR-CSA', 'MSCA-COFUND-FP', 'IA', 'MSCA-IF-GF')
	SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects where \`ec_fund_scheme\` in ('CP', 'ERC-AG')
		0.4657362103

ec_fund_scheme	framework_program	ec_max_contribution	total_cost
IA	H2020	8186834.01	9127173.01
IA	H2020	16704977.14	23370548.91
MSCA-ITN-ETN	H2020	3939999.12	3939999.12
RIA	H2020	5982101.25	5988163.75
CSA	H2020	2999405.0	2999405.0
IA	H2020	21009064.67	28419785.01
RIA	H2020	5673272.5	5927428.75

SELECT total_cost,ec_max_contribution,framework_program,ec_fund_scheme FROM projects where \`ec_fund_scheme\` in ('IMI2-CSA', 'MC-IOF', 'JTI-CP-ENIAC', 'CP-IP', 'CSA-ERA-Plus', 'JTI-CS', 'JTI-CSA-FCH', 'ERC-COG', 'CP-CSA-Infra', 'JTI-CP-FCH', 'MC-ITN', 'CSA-LS', 'MC-IRG', 'MSCA-COFUND-FP', 'MSCA-ITN-ETN', 'Shift2Rail-RIA', 'MC-IIF', 'BBI-CSA', 'Shift2Rail-IA', 'CP-TP', 'CP-IP-SICA', 'NoE', 'SGA-CSA', 'BBI-IA-DEMO', 'SGA-RIA', 'BBI-RIA', 'CP-SICA', 'ERC-STG', 'ERC-CG', 'JTI-CP-IMI', 'MSCA-IF-GF', 'MSCA-ITN-EID', 'ECSEL-CSA', 'RIA-LS', 'SESAR-IA', 'MSCA-IF-EF-SE', 'ERC-SG', '0.0', 'CS2-CSA', 'MC-IRSES', 'CS2-IA', 'Shift2Rail-RIA-LS', 'CSA-SA', 'PCP', 'MSCA-RISE', 'CP', 'MSCA-IF-EF-CAR', 'BSG-SME', 'IA-LS', 'CIP-EIP-EI-PMRP', 'BPN', 'SME-1', 'CP-FP', 'ERC-POC', 'H2020-EEN-SGA', 'BBI-IA-FLAG', 'MC-CIG', 'CP-CSA-Infra-PP', 'CSA-ERANET', 'CSA', 'CSA-SA(POC)', 'CSA-CA', 'CS2-RIA', 'MC-IAPP', 'PA', 'ERC-SyG', 'BSG-CSO', 'MC-COFUND', 'MC-IIFR', 'SME-2', 'Shift2Rail-IA-LS', 'MSCA-COFUND-DP', 'COFUND-PCP', 'FCH2-RIA', 'IA', 'MC-ERG', 'FCH2-CSA', 'ERC-ADG', 'SESAR-CSA', 'ECSEL-RIA', 'SESAR-RIA', 'PPI', 'FCH2-IA', 'COFUND-PPI', 'IMI2-RIA', 'MSCA-ITN-EJD', 'PB', 'MC-IEF', 'CPCSA', 'Shift2Rail-CSA', 'ERC-LVG', 'TN', 'ERC-AG', 'ERC-POC-LS', 'RIA', 'ECSEL-IA', 'JTI-CP-ARTEMIS', 'MSCA-IF-EF-RI', 'CP-FP-SICA', 'BSG-SME-AG', 'COFUND-EJP', 'CP-SoU', 'ERA-NET-Cofund', 'CP-CSA') and \`framework_program\` in ('H2020')	cordis	Run Query

Figure 1: Example with the CORDIS data set.



SELECT \`objid\`,\`ra\`,\`dec\` FROM photoprimary LIMIT 10000

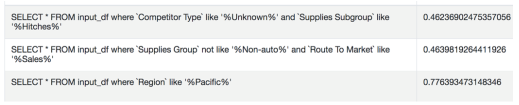
	objid	ra	dec
objid	1.00	0.44	0.91
ra	0.44	1.00	0.41
dec	0.91	0.41	1.00

View	Query	Scores	Action

['ra']	SELECT \`objid\`,\`ra\`,\`dec\` FROM photoprimary where \`ra\` < 184.50983
	SELECT \`objid\`,\`ra\`,\`dec\` FROM photoprimary where \`ra\` < 185.55168 and \`ra\` >= 184.50983
	SELECT \`objid\`,\`ra\`,\`dec\` FROM photoprimary where \`ra\` >= 185.55168
		0.9394572377204895	Execute Query
			Execute Query
			Execute Query

['objid' 'dec']	SELECT \`objid\`,\`ra\`,\`dec\` FROM photoprimary where \`dec\` < -0.9436114
	SELECT \`objid\`,\`ra\`,\`dec\` FROM photoprimary where \`dec\` >= -0.9436114
		0.9187800884246826	Execute Query
			Execute Query

Figure 2: Example with the SDSS data set.



SELECT * FROM input_df where \`Competitor Type\` like '%Unknown%' and \`Supplies Subgroup\` like '%Hitches%'	0.46236902475357056
SELECT * FROM input_df where \`Supplies Group\` not like '%Non-auto%' and \`Route To Market\` like '%Sales%'	0.4639819264411926
SELECT * FROM input_df where \`Region\` like '%Pacific%'	0.776393473148346

Figure 3: Example with textual data on the IBM data set.

to show how query recommendations work, and show a variety of interesting examples. Participants will be able to try recommendations on different data sets and 'play' with the algorithm parameters to receive different recommendations.

Demo Examples. Example cases include:

Example 1: We start the exploration of the PhotoPrimary table of the SDSS dataset. This dataset is entirely numeric. Our aim is to find interesting regions in the sky. Interestingly enough, PyExplore suggests a region that is close to the one provided in one of the queries of the SDSS documentation[11]. In Figure 2, we can see the correlation matrix between the attributes and the ability for the user to execute the queries provided by the recommendation system.

Example 2: We start with a query on the CORDIS dataset. In this case, we will see how PyExplore handles categorical and numerical attributes in a real-life data set. In Figure 1, we see some recommendations on *ec_fund_scheme* and *framework_program*, which are categorical, and *ec_max_contribution*, which is numerical.

Example 3: On the IBM Sales data set, we will use the String workflow. Clicking the button "String Options", we can modify the workflow parameters. We will run two variations one using TF-IDF for dataset vectorization and one using doc2vec, and compare the results. In Figure 3, we see some example recommendations produced by PyExplore using the textual attributes *Competitor Type*, *Supplies Group*, and *Region*. We can see that PyExplore suggests interesting values such as "Pacific" for *Region*.

Example 4: PyExplore offers a large number of opportunities to experiment and evaluate recommendations. For example, we will compare recommendations based on correlations vs diversity, choose different clustering algorithms e.g., using OPTICS instead of the hierarchical clustering to exclude non-correlated attributes, see the impact of doc2vec in the quality of recommendations for textual data, and so forth.

[11]http://skyserver.sdss.org/dr8/en/help/docs/realquery.asp

ACKNOWLEDGMENTS

This work has been partially funded by the European Union's Horizon 2020 research and innovation program (grant agreement No 863410)

REFERENCES

[1] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: ordering points to identify the clustering structure. *ACM Sigmod record* 28, 2 (1999), 49–60.
[2] Harald Cramér. 1999. *Mathematical methods of statistics*. Vol. 43. Princeton university press.
[3] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In *ACM SIGKDD*. 71–80.

[4] Magdalini Eirinaki and Sweta Patel. 2015. QueRIE reloaded: Using matrix factorization to improve database query recommendations. In *IEEE Big Data*. 1500–1508.

[5] Bill Howe, Garrett Cole, Nodira Khoussainova, and Leilani Battle. 2011. Automatic example queries for ad hoc databases. In *ACM SIGMOD*. ACM, 1319–1322.

[6] Zhexue Huang. 1997. Clustering large data sets with mixed numeric and categorical values. In *PAKDD*. Singapore, 21–34.

[7] Zhexue Huang. 1997. A fast clustering algorithm to cluster very large categorical data sets in data mining. *DMKD* 3, 8 (1997), 34–39.

[8] Zhexue Huang. 1998. Extensions to the k-means algorithm for clustering large data sets with categorical values. *Data mining and knowledge discovery* 2, 3 (1998), 283–304.

[9] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. 2010. SnipSuggest: Context-Aware Autocompletion for SQL. *Proc. VLDB Endow.* 4, 1 (2010), 22–33.

[10] Quoc Le and Tomas Mikolov. 2014. Distributed representations of sentences and documents. In *International conference on machine learning*. 1188–1196.

[11] Marie Le Guilly, Jean-Marc Petit, Vasile-Marian Scuturici, and Ihab F Ilyas. 2019. ExplIQuE: Interactive Databases Exploration with SQL. In *ACM CIKM*. 2877–2880.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*. 3111–3119.

[13] Tova Milo and Amit Somech. 2020. Automating Exploratory Data Analysis via Machine Learning: An Overview. In *SIGMOD Conference 2020*. ACM, 2617–2622.

[14] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[15] M. Qiu. 2004. Evaluation of Clustering Techniques In Data Mining Tools.

[16] David Sculley. 2010. Web-scale k-means clustering. In *World Wide Web Conference*. 1177–1178.

[17] Thibault Sellam and Martin Kersten. 2016. Cluster-driven navigation of the query space. *IEEE TKDE* 28, 5 (2016), 1118–1131.

[18] Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal* 27, 3 (1948), 379–423.

[19] Xiaoyan Yang, Cecilia M. Procopiuc, and Divesh Srivastava. 2009. Recommending Join Queries via Query Log Analysis. In *ICDE*. 964–975.