Semantic Evaluation for Text-to-SQL with Distilled Test Suite

Anonymous EMNLP submission

Abstract

We propose *test suite accuracy* to approximate semantic accuracy for Text-to-SQL models, where a predicted query is semantically correct if its denotation is the same as the gold for every possible database. Our method distills a small test suite of databases that achieves high code coverage for the gold query from a large number of randomly generated databases. At evaluation time, it computes the denotation accuracy of the predicted queries on the distilled test suite, hence calculating a tight upper-bound for semantic accuracy efficiently. We generate a distilled test suite for SPIDER, COSQL, and SPARC, and evaluate 21 models submitted to the SPIDER leaderboard. We manually examine 100 predictions where our approach disagrees with the current metric, and verify that our method is always correct. The current metric of SPIDER leads to a 2.5% false negative rate on average and 8.1% in the worst case, indicating that test suite accuracy is needed to reflect progress in semantic parsing better.

1 Introduction

A Text-to-SQL model translates natural language instructions to SQL queries that can be executed on databases and bridges the gap between expert programmers and non-experts. Accordingly, researchers have built a diversity of datasets (Dahl, 1989; Iyer et al., 2017; Zhong et al., 2017; Yu et al., 2018) and improved model performances (Xu et al., 2017; Suhr et al., 2018; Guo et al., 2019; Bogin et al., 2019a; Wang et al., 2020). However, evaluating the semantic accuracy of a Text-to-SQL model is a long-standing problem: we want to know whether the predicted SQL query has the same denotation as the gold for all every possible database. "Single" denotation evaluation executes the predicted SQL on one database and compares its denotation with that of the gold. It might create *false positives*, where a semantically different prediction (Figure 1 prediction 1) happens to have the same denotation as the gold, on a particular database. In contrast, exact string match might produce *false negatives*: Figure 1 prediction 2 is semantically equivalent to the gold but differs in logical form.

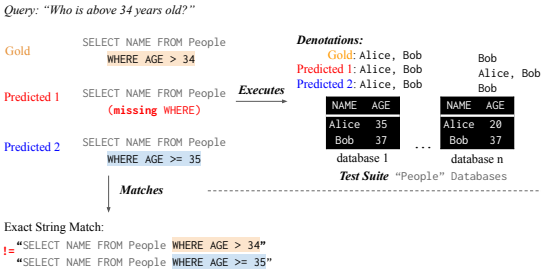


Figure 1: Prediction 2 is semantically correct, and Prediction 1 is wrong. Exact string match judges prediction 2 to be wrong, which leads to *false negatives*. Only comparing denotations on database 1 judges prediction 1 to be correct, which leads to *false positives*. Test suite evaluation compares denotations on a set of databases and reduces false positives.

The programming language research community developed formal tools to reliably reason about query equivalence for a restricted set of query types. They lift SQL queries into other semantic representations such as K-relations (Green et al., 2007), UniNomial (Chu et al., 2017) and U-semiring (Chu et al., 2018); then they search for an equivalence or inequivalence proof. However, these representations cannot express sort operations and float comparisons, and hence do not support the full range of operations that Text-to-SQL models can use. We ideally need a method to approximate semantic accuracy reliably without operation constraints.

If the computation budget were unlimited, we could compare the denotations of the predicted query with those of the gold on a large number of

random databases (Section 4.1), and obtain a tighter upper bound for semantic accuracy than single denotation evaluation. The software testing literature calls this idea fuzzing (Padhye et al., 2019; AFL; Lemieux et al., 2018; Qui). However, it is undesirable to spend a lot of computational resources *every time* when we evaluate a Text-to-SQL model. Instead, we want to check denotation correctness only on a smaller test suite of databases that are more likely to distinguish [1] *any* wrong model predictions from the gold.

Inspired by fuzzing, we propose *test suite accuracy* to efficiently approximate the semantic accuracy of a Text-to-SQL model, by checking denotations of the predicted queries on a compact test suite with high code coverage. We introduce how to search for such a test suite without prior knowledge about model predictions. In Section 3.1, we generate *neighbor queries* of a gold program by modifying only one aspect of it. For example, prediction 1 in Figure 1 is a neighbor query of the gold, since they differ only by a filtering predicate. Neighbor queries have two desirable properties: (1) they are usually semantically different from the gold, and (2) if a test suite can distinguish them from the gold, it is likely to distinguish other wrong predictions as well. The latter holds because distinguishing all neighbors from the gold requires executions on these databases to exercise every modified part of the gold query, hence reflecting comprehensive code coverage and high test quality (Miller and Maloney, 1963; Ammann and Offutt).

We generate a large number of random databases and keep a small fraction of them that can distinguish the neighbors from the gold (Section 4.2). We call this set of databases a *distilled test suite*. While evaluating model predictions, we only check their denotations on the distilled test suite to approximate semantic accuracy efficiently.

1.1 Application

We construct distilled test suites for SPIDER (Yu et al., 2018), COSQL (Yu et al., 2019a) and SPARC (Yu et al., 2019b); on all three datasets, the test suites can distinguish more than 99% of the neighbor queries after generating 1000 random databases. We evaluate test suite accuracy on 21 SPIDER leaderboard submissions, randomly sample 100 model predictions where our method disagrees with exact set match (ESM, the SPIDER official metric), and manually verify that our method is correct in *all* these cases (Section 6.1).

We use test suite accuracy as a proxy for semantic accuracy to examine how well the exact set match metric approximates the semantic accuracy, and identify several concerns. (1) The exact set match (ESM) tends to underestimate model performances, leading to a 2.5% false negative rate on average and 8.1% in the worst case. (2) ESM does not reflect all improvements in semantic accuracy. For example, it undervalues a high-score submission with 61% semantic accuracy by 8%, but instead favors five other submissions with lower semantic accuracy, thus misrepresenting state of the art. (3) ESM becomes poorer at approximating semantic accuracy on more complex queries. Since models are improving and solving harder queries, ESM deviates more from semantic accuracy. We need test suite accuracy to better track progress in Text-to-SQL development.

Our contributions are summarized as follows:

- A method and software to create compact high quality test suites for Text-to-SQL evaluation.
- A test suite to reliably approximate semantic accuracy for SPIDER, COSQL and SPARC.
- A detailed analysis of why current metrics are poor at approximating semantic accuracy.

2 Problem Statement

Let $w \in \mathcal{W}$ be a database input to a SQL query $q \in Q$, and $[\![q]\!]_w$ be the denotation of q on w ,[2] where $\mathcal{W}/Q$ is the space of all databases/SQL queries. Two queries q_1 and q_2 are semantically equivalent if their denotations are the same for all possible databases, i.e.

$$\forall w \in \mathcal{W}, [\![q_1]\!]_w = [\![q_2]\!]_w \quad (1)$$

We refer to the ground truth query as g and the predicted query to be evaluated as q. Ideally, we want to evaluate whether g and q are semantically equivalent (abbreviated as **semantic accuracy**), which is unfortunately undecidable in general (Chu et al., 2017). Traditionally, people evaluate a model prediction q by either **exact string match** or compare denotations on a single database w (abbreviated as **single denotation accuracy**). Exact string

[1] Section 2 defines that a database distinguishes two queries if their executions lead to different results.

[2] As in Yu et al. (2018), we use Sqlite3 to obtain the denotation. Define $[\![q]\!]_w = \perp$ if execution does not end, which is implemented as timeout in practice.

match is too strict, as two different strings can have the same semantics. Single denotation evaluation is too loose, as the denotations of g and q might be different on another database w.

We use **test suite** to refer to a set of databases. A database w **distinguishes** two SQL queries g, q if $[\![g]\!]_w \neq [\![q]\!]_w$, and a test suite S distinguishes them if one of the databases $w \in S$ distinguishes them:

$$\exists w \in S, [\![g]\!]_w \neq [\![q]\!]_w \tag{2}$$

For convenience, we define the indicator function:

$$D_S(g, q) \coloneqq \mathbb{1}[S \text{ distinguishes } g, q] \tag{3}$$

We use the test suite S to evaluate a model prediction q: q is correct iff $D_S(g, q) = 0$; i.e., g and q have the same denotations on all databases in test suite S. Sorted by looseness, exact match $<$ semantic accuracy $<$ test suite accuracy $<$ single denotation accuracy. Our goal is to find a test suite of databases S, such that it can be used to approximate semantic accuracy reliably and efficiently.

3 Desiderata

We use S_g to denote the target test suite. Before describing how to generate S_g, we first list two criteria of a desirable test suite. Later we construct S_g by optimizing over these two criteria.

Code coverage. The test suite needs to cover every branch and clauses of the gold program such that it can test the use of every crucial clause, variable, and value. For example, database 1 in Figure 1 alone does not have a row where "AGE $\leq$ 34" and hence does not have comprehensive code coverage.

Computational efficiency. We minimize the size of S_g to speed up test suite evaluations.

3.1 Neighbor Queries

We measure the code coverage of a test suite by its ability to distinguish the gold query from its *neighbor queries* that are likely to be semantically different but close in surface forms. To generate them, we modify one of the following aspects of the gold query (Figure 2): (1) replace an integer (float) value with either a random integer (float) or its value $\pm$ 1 (0.001); (2) replace a string with a random string, its sub-string or a concatenation of it with another random string; (3) replace a comparison operator/column name with another; (4) drop a non-optional span (e.g., the default sort order "ASC" is optional). We then remove modified queries that cannot compile and execute.

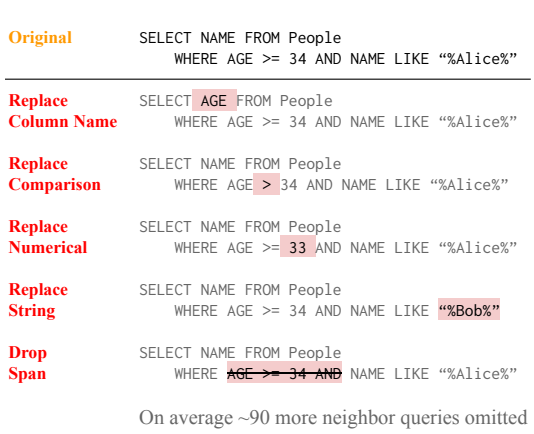


Figure 2: Automatically generating a set of neighbor queries N_g. We apply one type of modification to the original query at a time. The modified queries are likely to be semantically close but inequivalent to the gold.

Neighbor queries have two desirable properties. First, they are likely to be semantically different from the gold query. For example, " $>$ 34" is semantically different from " $\geq$ 35" (replace comparison operator) and " $>$ 35" (replace values); however, we only apply one modification at a time, since " $>$ 34" is semantically equivalent to " $\geq$ 35" for an integer. Secondly, in order to distinguish the gold from all its neighbors, the test suite needs to cover all the branches of the gold program. For example, the database needs to have people above, below and equal to age 34 to distinguish all is neighbors. Hence, the test suite tends to have high quality if it can distinguish the gold from all its neighbors.

Our goal is to find a *small* test suite that can distinguish as *many* neighbor queries as possible. Denoting the set of neighbors for the gold program g as N_g, we hope to find a test suite S_g:

$$\begin{aligned} &\text{minimize } |S_g| \\ s.t. \quad &\forall q \in N_g, D_{S_g(g,q)} = 1 \end{aligned} \tag{4}$$

4 Fuzzing

Fuzzing is a software testing technique that generates a large number of random inputs to test whether a program satisfies the target property (e.g., SQL equivalence). We describe a procedure to sample a large number of random databases and keep a small fraction of them to distill a test suite S_g.

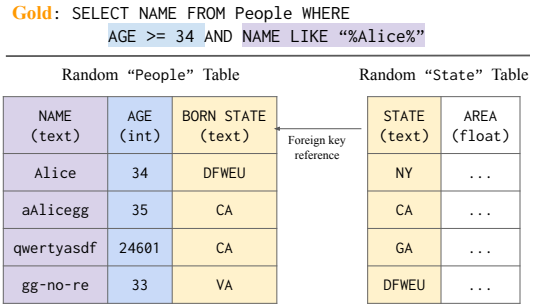


Figure 3: A random database input w from the distribution $\mathcal{I}_g$, where g is the gold SQL query. We generate the "State" column before the "BORN STATE" column because the latter has to be a subset of the former. Each element of the column "BORN STATE" is sampled uniformly at random from the parent "STATE" column. For the column that has data type int/string, each element is either a random number/string or a close variant of the values in the gold query.

4.1 Sampling Databases

A database w needs to satisfy the input type constraints of the gold program g, which include using specific table/column names, foreign key reference structure, and column data types. We describe how to generate a random database under these constraints and illustrate it with Figure 3.

If a column c_1 refers to another column c_2 as its foreign key, all elements in c_1 must be in c_2 and we have to generate c_2 first. We define a partial order among the tables: table $A <$ table B if B has a foreign key referring to any column in table A. We then generate the content for each table in descending order found by topological sort. For example, in Figure 3, we generate the "State" table before the "People" table because the latter refers to the former. We now sample elements for each column such that they satisfy the type and foreign key constraints. If a column c_1 is referring to another column c_2, each element in c_1 is uniformly sampled from c_2. Otherwise, if the column is a numerical(string) type, each element is sampled uniformly from $[-2^{63}, 2^{63}]$ (a random string distribution). We also randomly add in constant values used in g (e.g., 34 and "Alice") and their close variants (e.g., 35 and "aAlicegg") to potentially increase code coverage. We denote the database distribution generated by this procedure as $\mathcal{I}_g$.

4.2 Distilling a Test Suite

We use samples from $\mathcal{I}_g$ to construct a small test suite S_g such that it can distinguish as many neighbor queries in N_g as possible (Section 3.1). We initialize S_g to be empty and proceed greedily. A database w is sampled from the distribution $\mathcal{I}_g$; if w can distinguish a neighbor query that cannot be distinguished by any databases in S_g, we add w to S_g. Appendix Section A.1 gives a more rigorous description. In the actual implementation, we also save the disk space by sharing the same random database w_t across all gold SQL queries that are associated with the same schema. Though this algorithm is far from finding an optimal solution to Objective 4, in practice, we find a test suite that is small enough to distinguish most neighbor queries.

5 Data

Datasets We generate test suites S_g for the development set of 3 datasets: SPIDER (Yu et al., 2018), SPARC (Yu et al., 2019b) and COSQL (Yu et al., 2019a). Since they share the same set of database schema, we generate the same set of databases for them to save space.

Model Predictions We run test suite evaluation on real model predictions for the SPIDER development set, which contains 1034 language-SQL pairs. It stratifies data into four categories (easy, medium, hard, and extrahard) according to difficulty level measured by gold SQL complexity. We decide to focus on SPIDER because it invites researchers to submit their model predictions and requires them to follow a standard format, which makes it convenient to study a wide variety of model predictions.

We obtained the development set model predictions from 21 submissions. They include models from Guo et al. (2019); Bogin et al. (2019b); Choi et al. (2020); Wang et al. (2020). [3] These models capture a broad diversity of network architectures, decoding strategies, and pre-traning methods, with accuracy (defined later) ranging from below 40% to above 70%. We obtained the model predictions *after* producing the test suites to ensure that our method is general and not tailored to a specific family of model predictions. To foster reproducibility, we obtain consent from the corresponding author of Yu et al. (2018) to release the model predictions.

Metric Adaptation The SPIDER official evaluation metric is exact set match (Zhong et al., 2017; Yu et al., 2018). It parses the gold and predicted SQLs into sub-clauses and determine accuracy by checking whether they have the same *set* of clauses.

[3] Many dev set submissions do not have public references.

It improves over exact string matching by preventing false negatives due to semantically equivalent clause reordering. However, it is still considered to be a strict metric and create false negatives.

We adapt both test suite accuracy and SPIDER official exact set match (abbreviated as ESM) to make a fair comparison. Because ESM does not account for value prediction correctness, we enumerate all possible ways to replace the values in a predicted query with the gold values, and consider a prediction to be correct if one of the replacements passes the test suite. ESM is also indifferent towards column order differences, so we consider two denotations equivalent if they only differ by a column permutation.

The official SPIDER evaluation script accidentally ignores any join predicate (Figure 8 row 1). We fix this issue before comparing ESM to test suite accuracy. We always refer to these adapted metrics rather than the original ones unless we explicitly specify.

6 Results

The test suite S_g for all three datasets are shared and takes 3.27GB in space (databases from the original datasets take 100.7MB). Table 1 reports the time needed to evaluate on the entire test suite. Although test suite evaluation consumes more space and computation resources than single denotation evaluation, it is parallelizable and affordable by most researchers.

Dataset	# Queries	One (min)	Suite (min)
SPIDER	1034	1.2	75.3
COSQL	1007	1.1	75.6
SPARC	1203	1.4	86.7

Table 1: Development set sizes, the wall clock time needed (on one CPU) to execute the gold query only on databases provided by (Yu et al., 2018) (One), and the time needed to run on the entire test suite (Suite).

6.1 Reliability

Distinguish Neighbor Queries. For each gold query in SPIDER/COSQL/SPARC, we generate on average 94/93/81 neighbor queries (Figure 2). We sample 1000 random databases for each database schema and run fuzzing (Section 4.2) to construct S_g, which takes around a week on 16 CPUs. Figure 5 reveals the progress of fuzzing by plotting the fraction of neighbor queries that remain undistinguished after attempting t random databases.

Checking single database denotation fails to distinguish 5% of the neighbor queries, and the curve stops decreasing after around 600 random databases. For all three datasets, 1000 random databases can distinguish $> 99\%$ of the neighbor queries. A large number of random databases is necessary to achieve comprehensive code coverage.

Figure 4 presents some typical neighbor queries that have the same denotations as the gold on all the databases we sampled. These queries are only a small fraction (1%) of all the neighbors; in most cases they happen to be semantically equivalent to the gold. We acknowledge that our fuzzing based approach has trouble distinguishing semantically close queries that differ only at a floating-point precision (e.g. "≤ 2.31" vs. "< 2.31"). Fortunately, however, we cannot find a false positive caused by this weakness in our subsequent manual evaluation.

Manual evaluation. Even though our test suite achieves comprehensive code coverage, we still need to make sure that our method does not create any false positives on real model predictions. We focus on the predictions from the 21 submissions that are considered incorrect by ESM but correct by our test suite evaluation, and manually examined 100 of them. *All* of them are semantically equivalent to the gold query; in other words, we did not observe a single error made by our evaluation method. We will release these 100 model predictions along with annotated reasons for why they are equivalent to the gold labels, such that the research community can conveniently scrutinize the quality of our evaluation method.

Difficulty	Mean	Std	Max
easy (%)	0.5 / 2.2	0.5 / 1.3	2.0 / 7.2
medium (%)	0.2 / 1.9	0.3 / 1.9	0.7 / 8.0
hard (%)	0.5 / 4.4	1.2 / 3.8	4.0 / 12.1
extra (%)	1.7 / 3.2	1.8 / 1.6	5.3 / 8.2
all data (%)	0.5 / 2.6	1.0 / 1.7	2.0 / 8.1

Table 2: The false positive/negative rate of the adapted exact set match for each difficulty split in the SPIDER dataset. We report the mean/standard deviation/max of these two statistics among 21 dev set submissions.

6.2 Errors of Traditional Metrics

Given that test suite evaluation empirically provides an improved approximation of semantic equivalence, we use test suite accuracy as ground truth and retrospectively examine how well ESM

Modification	Gold & Modified	Passing Reason
Comparison Operator Replaced	Gold: SELECT T1.NAME FROM Conductor … GROUP BY T2.CONDUCTER_ID HAVING COUNT(*) > 1 Modified: SELECT T1.NAME FROM Conductor … GROUP BY T2.CONDUCTER_ID HAVING COUNT(*) != 1	Count is always positive, so "> 1" is equivalent to "!= 1". modification is semantically equivalent to the original SQL.
Constant Replaced	Gold: SELECT NAME FROM City WHERE POPULATION BETWEEN 160000 AND 90000 Modified: SELECT NAME FROM City WHERE POPULATION BETWEEN 160000 AND 21687	Original annotation is wrong and both the original and the modification lead to empty results, which are semantically equivalent.
Column Name Dropped	Gold: SELECT COUNT(T2.LANGUAGE), T1.NAME … Modified: SELECT COUNT(*), T1.NAME FROM …	The SQL interpreter infers it should count the number of rows. modification is semantically equivalent to the original SQL.
Comparison Operator Replaced	Gold: SELECT COUNT(*) FROM Dogs WHERE age < (SELECT AVG(AGE) FROM Dogs) Modified: SELECT COUNT(*) FROM Dogs WHERE age <= (SELECT AVG(AGE) FROM Dogs)	A dog entry needs to have exactly the average age to distinguish the modification. This happens with low probability and our test suite fails to distinguish them.

Figure 4: Representative modifications in N_g that produce the same results as the gold (pass) on all sampled databases.

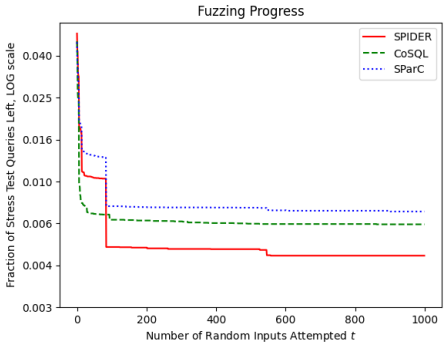


Figure 5: The progress of fuzzing (Section 4.2). Each curve represents a different dataset. The x-axis is the number of random databases attempted (t), and the y-axis (re-scaled by log) is the fraction of neighbor queries left. y-value at $x = 0$ is the fraction of neighbors left after checking denotations on the database provided by Yu et al. (2018). Figure 4 shows representative remaining neighbors when fuzzing finishes.

approximates semantic accuracy. We calculate the false positive/false negative rate for each difficulty split and report the mean, standard deviation, and max for all 21 model submissions. Table 2 shows the results. ESM leads to a nontrivial false negative rate of 2.6% on average, and 8.1% in the worst case. The error becomes larger for harder fractions of queries characterized by more complex queries. On the hard fraction, false negative rate increases to 4% on average and 12.1% in the worst case.

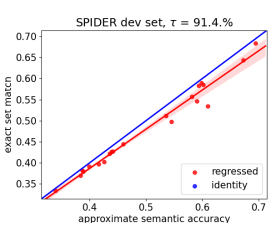
In Table 3, we report the difference between test suite accuracy and single denotation accuracy, which effectively means testing the predicted SQL query only on the databases from the original dataset release (Yu et al., 2018). In the worst case, single denotation accuracy creates a false positive rate of 8% on the entire development set, and 4% more on the extrahard fraction.

Difficulty	Mean	Std	Max
easy (%)	3.6	1.2	6.0
medium (%)	5.9	0.9	8.2
hard(%)	8.0	1.5	10.3
extra (%)	11.0	3.5	17.6
all data (%)	6.5	1.0	9.0

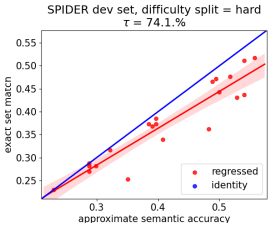
Table 3: The false positive rate of single denotation accuracy (i.e., checking denotation only on the database originally released in Yu et al. (2018)) for each difficulty split of the SPIDER dataset. We report the mean/standard deviation/max of these two statistics among 21 dev set submissions.

6.3 Correlation with Existing Metrics

Could surface-form based metric like ESM reliably track improvements in semantic accuracy? We plot ESM against test suite accuracy for all 21 dev set predictions in Figure 6. On a macro level, ESM correlates well with test suite accuracy with Kendall τ correlation 91.4% in aggregate; however, the correlation decreases to 74.1% on the hard fraction. Additionally, ESM and test suite accuracy starts to diverge as model performance increases. These two facts jointly imply that as models are be-

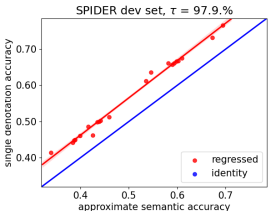


(a) $\tau = 91.4\%$ on all queries in the dev set.

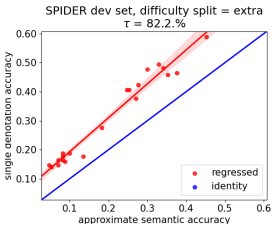


(b) $\tau = 74.1\%$ on hard fraction of the dev set.

Figure 6: Kendall τ correlation between exact set match and test suite accuracy. Each dot is a development set submission to the SPIDER leaderboard.



(a) $\tau = 97.9\%$ on all queries in the dev set.



(b) $\tau = 82.2\%$ on extrahard fraction of the dev set.

Figure 7: Kendall τ correlation between single execution accuracy as originally defined in Yu et al. (2018) and test suite accuracy. Each dot is a dev set submission to the SPIDER leaderboard.

coming better at harder queries, ESM is no longer sufficient to approximate semantic accuracy. On a micro level, when two models have close performances, improvements in semantic accuracy might not be reflected by increases in ESM. On the hard fraction, 5 out of 21 submissions have more than four others that have lower test suite accuracy but higher ESM scores (i.e., five dots in Figure 6b have four dots to their upper left).

Figure 7 plots the correlation between single denotation accuracy against test suite accuracy. On the extrahard fraction, four submissions have more than three others that have higher single denotation accuracy but lower test suite accuracy. Checking denotation only on one database is insufficient.

We list the Kendall τ correlations between test suite accuracy and different metrics in Table 4 and plot them in the appendix Section A.2. The correlation with the current official metric is low without fixing the issue identified at the end of Section 5.

7 Metrics Comparison and Analysis

We explain how ESM and test suite accuracy differ and provide representative examples.

Difficulty	Adpated	Official	Single Denot
easy (%)	91	86	90
medium (%)	90	37	96
hard (%)	75	28	94
extra (%)	91	20	82
all data (%)	91	40	98

Table 4: Kendall τ correlation between various metrics and test suite accuracy across 21 model prediction files. **Adapted** refers to ESM after we fix the issue identified at the end of Section 5. **Official** refers to directly running the official evaluation script to evaluate, and **Single Denot** refers to only checking denotation on the one database provided by Yu et al. (2018).

7.1 False Positives

Although ESM is usually considered strict, the SPIDER evaluation script ignores `JOIN` predicates and leads to false positives. Additionally, since multiple intermediate tables can contain the same column, selecting any of them is semantically equivalent. Yu et al. (2018) addressed this problem by not evaluating intermediate table names. Such a strategy effectively rules out many false negatives but also introduces new false positives. Figure 8 row 1 shows an example where the "`JOIN`" predicate is missing, and ESM ignores table name differences.

7.2 False Negatives

We provide representative false negative mistakes made by ESM in Figure 8 row 2-7. As we can see from row 2-4, slightly complicated queries usually have semantically equivalent variants, and it is nontrivial to tell whether they are semantically equivalent unless we execute them on a test suite or manually verify them.

Nevertheless, even though test suite accuracy reliably approximates semantic accuracy according to our observation, researchers might also care about other aspects of a generated SQL query. Semantic accuracy is only concerned with *what* are the denotations of a query, but not *how* it calculates them. For example, Figure 8 row 5 represents one of the most common types of false negatives, where the predicted SQL query chooses to join other tables even though it is unnecessary. While semantically correct, the predicted query increases running time. Figure 8 row 7 exhibits a similar but more complicated and rare example.

Inserting gold values into model predictions as described in Section 5 might also unexpectedly loosen the semantic accuracy metric. For exam-

	Error	Gold & Model Prediction	Explanation
1	False Positive	**Gold:** SELECT **T3.**NAME, T2.COURSE FROM Course_arrange AS T1 JOIN Course AS T2 **ON** **T1.COURSE_ID = T2.COURSE_ID** JOIN Teacher AS T3 **ON T1.TEACHER_ID = T3.TEACHER_ID**; **Prediction:** SELECT **T1.**NAME, T2.COURSE FROM Teacher AS T1 JOIN Course_arrange AS T3 JOIN Course AS T2; [Missing JOIN keys]	Exact set match does not account for predicates used by a JOIN clause; it also ignores variable names.
2	False Negative	**Gold:** SELECT TEMPLATE_ID FROM Templates **EXCEPT** SELECT TEMPLATE_ID FROM Documents; **Prediction:** SELECT TEMPLATE_ID FROM Templates **WHERE TEMPLATE_ID NOT IN** (SELECT TEMPLATE_ID FROM Documents);	"EXCEPT" is semantically equivalent to "NOT IN"
3	False Negative	**Gold:** SELECT COUNT(*) FROM Area_code_state; **Prediction:** SELECT COUNT(**STATE**) FROM Area_code_state;	Counting any column is the same.
4	False Negative	**Gold:** SELECT TRANSCRIPT_DATE FROM Transcripts **ORDER BY TRANSCRIPT_DATE DESC LIMIT 1**; **Prediction:** SELECT **MAX**(TRANSCRIPT_DATE) FROM Transcripts;	First element of descendingly sorted column is equivalent to maxing.
5	False Negative	**Gold:** SELECT COUNT(*) FROM Cars_data WHERE HORSEPOWER > 150; **Prediction:** SELECT COUNT(*) FROM Cars_data as T1 **JOIN Car_names as T2 on T1.ID = T2.MAKEID** where T1.HORSEPOWER > 150;	Semantically correct redundant join.
6	False Negative	**Gold:** SELECT AIRLINE FROM Airlines WHERE **ABBREVIATION = "UAL"**; **Prediction:** SELECT AIRLINE FROM Airlines WHERE **ABBREVIATION LIKE "UAL"**;	If the string value is the same, "=" is equivalent to "LIKE"
7	False Negative	**Gold:** SELECT LANGUAGE FROM Country_language GROUP BY LANGUAGE ORDER BY Count(*) DESC LIMIT 1; **Prediction:** SELECT **Country_language.**LANGUAGE FROM **Country JOIN** Country_language GROUP BY Country_language.LANGUAGE ORDER BY Count(*) Desc LIMIT 1;	The redundant join is implicitly a cross join, which will repeat every row in Country_language by [size of Country table] times. It leads to the same ranking if counted.

Figure 8: Representative examples where the exact set match (ESM) metric is different from test suite accuracy. False Positives happen when ESM judges a prediction to be correct but test suite accuracy judges it to be wrong; False Negatives happen when the reverse takes place.

ple, in Figure 8 row 6, the prediction uses the LIKE keyword rather than the "=" operator. By SQL style conventions, LIKE usually precedes a value of the form "%[name]%" and corresponds to natural language query "contains [name]" rather than "matches [name]"; it seems plausible that the model does not understand the natural language query. However, if we replace the wrong value "%[name]%" with the gold value "[name]" after the LIKE operator, the predicate becomes semantically equivalent to "= [value]" and hence makes the prediction correct. Value prediction is a crucial part of evaluating Text-to-SQL models.

8 Discussion and Conclusion

We propose test suite accuracy to approximate the semantic accuracy of a Text-to-SQL model, by automatically distilling a small test-suite with comprehensive code coverage from a large number of random inputs. We assure test suite quality by testing the test-suite with neighbor queries and manually examining its judgments on real model predictions. Our test suite will be released for SPIDER, SPARC and COSQL so that future works can conveniently evaluate test suite accuracy. This metric better reflects semantic accuracy, and we hope that it can inspire novel model designs and training objectives.

Our framework for creating test suites has two requirements: (1) the input is typed so that the fuzzing distribution $\mathcal{I}_g$ can be defined, and (2) slight modifications Q_g are semantically close but different from the gold g. Since these two conditions hold in many tasks, our framework might potentially be applied more broadly to other Text-to-SQL datasets (Zhong et al., 2017; Finegan-Dollak et al., 2018) and other logical forms, such as λ-DCS (Liang, 2013). We hope to see more future works that evaluate approximate semantic accuracy on the existing benchmarks and formulate new tasks amenable to test suite accuracy evaluation.

We do not attempt to solve SQL equivalence testing in general. While our test suite achieves comprehensive code coverage of the gold query, it might not cover all the branches of model predictions. Theoretically, we can always construct a query that differs from the gold only under extreme cases and fools our metric; however, we never observe models making such pathological mistakes.

Finally, as discussed in Section 7.2, there might be other crucial aspects of a predicted query beyond semantic correctness. Depending on the goal of the evaluation, other metrics such as memory/time efficiency and readability are also desirable and complementary to test suite accuracy.

References

American fuzzy lop. http://lcamtuf.coredump.cx/afl. Accessed: 2020-5-12.

Automatic testing of haskell programs. https://hackage.haskell.org/package/QuickCheck-2.14/docs/Test-QuickCheck.html. Accessed: 2020-5-12.

P Ammann and J Offutt. Introduction to software testing. cambridge university press, 2008.

Ben Bogin, Jonathan Berant, and Matt Gardner. 2019a. Representing schema structure with graph neural networks for text-to-sql parsing. In *ACL*.

Ben Bogin, Matt Gardner, and Jonathan Berant. 2019b. Representing schema structure with graph neural networks for text-to-sql parsing. *arXiv preprint arXiv:1905.06241*.

DongHyun Choi, Myeong Cheol Shin, EungGyun Kim, and Dong Ryeol Shin. 2020. Ryansql: Recursively applying sketch-based slot fillings for complex text-to-sql in cross-domain databases. *https://arxiv.org/abs/2004.03125*.

Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of sql queries. *Proceedings of the VLDB Endowment*, 11(11):1482–1495.

Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An automated prover for sql.

Deborah A. Dahl. 1989. Book reviews: Computer interpretation of natural language descriptions. *Computational Linguistics*, 15(1).

Catherine Finegan-Dollak, Jonathan K Kummerfeld, Li Zhang, Karthik Ramanathan, Sesh Sadasivam, Rui Zhang, and Dragomir Radev. 2018. Improving text-to-sql evaluation methodology. *arXiv preprint arXiv:1806.09029*.

Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40.

Jiaqi Guo, Zecheng Zhan, Yan Gao, Yan Xiao, Jian-Guang Lou, Ting Liu, and Dongmei Zhang. 2019. Towards complex text-to-sql in cross-domain database with intermediate representation. *arXiv preprint arXiv:1905.08205*.

Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, Jayant Krishnamurthy, and Luke Zettlemoyer. 2017. Learning a neural semantic parser from user feedback. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 963–973, Vancouver, Canada. Association for Computational Linguistics.

Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265.

Percy Liang. 2013. Lambda dependency-based compositional semantics. *arXiv preprint arXiv:1309.4408*.

Joan C Miller and Clifford J Maloney. 1963. Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63.

Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340.

Alane Suhr, Srinivasan Iyer, and Yoav Artzi. 2018. Learning to map context-dependent sentences to executable formal queries. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 2238–2249. Association for Computational Linguistics.

Bailin Wang, Richard Shin, Xiaodong Liu, Oleksandr Polozov, and Matthew Richardson. 2020. Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers.

Xiaojun Xu, Chang Liu, and Dawn Song. 2017. Sqlnet: Generating structured queries from natural language without reinforcement learning. *arXiv preprint arXiv:1711.04436*.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, Xi Victoria Lin, Yi Chern Tan, Tianze Shi, Zihan Li, Youxuan Jiang, Michihiro Yasunaga, Sungrok Shim, Tao Chen, Alexander Fabbri, Zifan Li, Luyao Chen, Yuwen Zhang, Shreya Dixit, Vincent Zhang, Caiming Xiong, Richard Socher, Walter Lasecki, and Dragomir Radev. 2019a. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 1962–1979, Hong Kong, China. Association for Computational Linguistics.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3911–3921.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, Suyi Li, Heyang Er, Irene Li, Bo Pang, Tao Chen, Emily Ji, Shreya Dixit,

David Proctor, Sungrok Shim, Jonathan Kraft, Vincent Zhang, Caiming Xiong, Richard Socher, and Dragomir Radev. 2019b. SParC: Cross-domain semantic parsing in context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4511–4523, Florence, Italy. Association for Computational Linguistics.

Victor Zhong, Caiming Xiong, and Richard Socher. 2017. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.

A Appendix

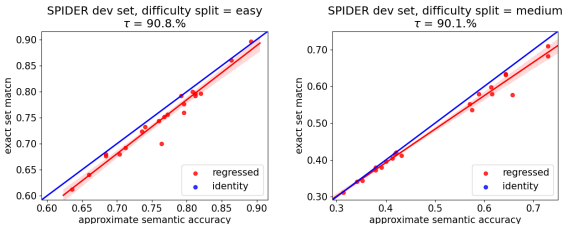
A.1 Algorithmic Description of Section 4.2

Algorithm 1: Distilling a test suite S_g. N_g is the set of neighbor queries of g; $\mathcal{I}_g$ is a distribution of database inputs.

$S_g := \emptyset, N := N_g$;
for $t = 1, 2, \ldots 1000$ **do**
  $w_t \sim \mathcal{I}_g$;
  for $q \in N_g$ **do**
    if $D_{\{w_t\}}(q, g) = 1$ **then**
      $S_g.add(w_t)$;
      $N.remove(q)$
return S_g

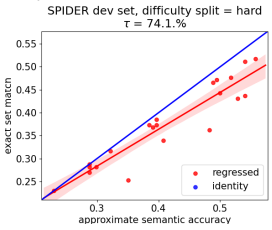
A.2 Correlation Plot with Other Metrics

We plot the correlation between test suite accuracy and (1) adapted exact set match (Figure 9), (2) official SPIDER exact set match (Figure 10), and (3) single denotation accuracy (Figure 11) on each fraction of the difficulty split.

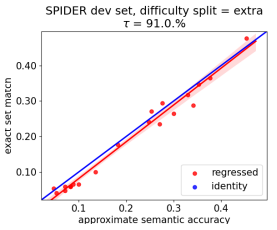


(a) $\tau = 90.8\%$ on easy fraction.

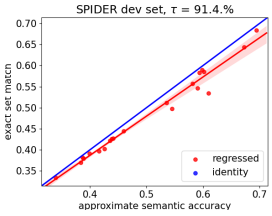
(b) $\tau = 90.1\%$ on medium fraction.



(c) $\tau = 74.1\%$ on hard fraction.

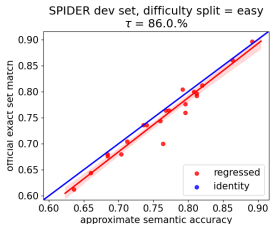


(d) $\tau = 91.0\%$ on extra hard fraction.

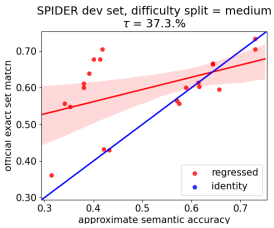


(e) $\tau = 91.4\%$ on all data.

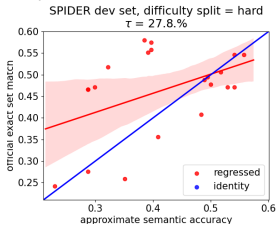
Figure 9: Kendall τ correlation between **adapted exact set match** and fuzzing-based accuracy. Each dot in the plot represents a development set submission to the SPIDER leaderboard.



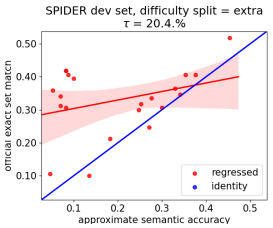
(a) $\tau = 86.0\%$ on easy fraction.



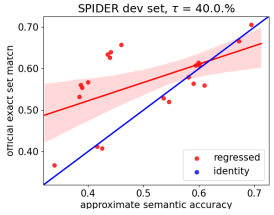
(b) $\tau = 37.3\%$ on medium fraction.



(c) $\tau = 27.8\%$ on hard fraction.

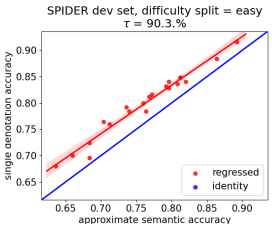


(d) $\tau = 20.4\%$ on extra hard fraction.

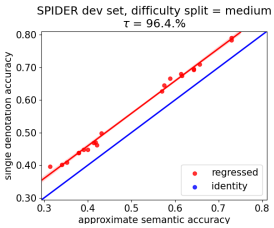


(e) $\tau = 40.0\%$ on all data.

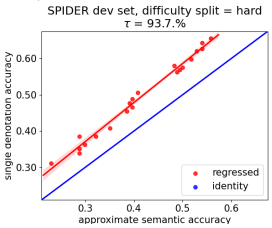
Figure 10: Kendall τ correlation between the official **SPIDER exact set match** and fuzzing-based accuracy. Each dot in the plot represents a development set submission to the SPIDER leaderboard.



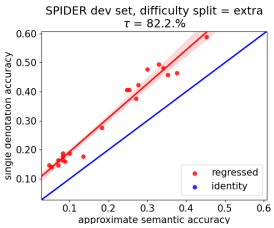
(a) $\tau = 90.3\%$ on easy fraction.



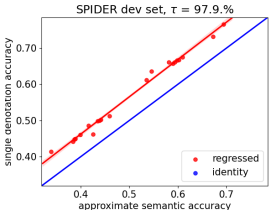
(b) $\tau = 96.4\%$ on medium fraction.



(c) $\tau = 93.7\%$ on hard fraction.



(d) $\tau = 82.2\%$ on extra hard fraction.



(e) $\tau = 97.9\%$ on all data.

Figure 11: Kendall τ correlation between **single denotation accuracy** and fuzzing-based accuracy. Each dot in the plot represents a development set submission to the SPIDER leaderboard.