MINO: Data-driven approximate type inference for Python

Stephen Tu
stephentu@csail.mit.edu
MIT CSAIL

1 INTRODUCTION

Dynamic programming languages, such as Python and Ruby, are becoming the languages of choice for many developers. The lack of strict compile-time type checking is often cited as making development cycles faster. However, a big drawback to dynamically typed languages is that it is impossible to know the exact type of a given variable at compile time; it is a well known result that this problem is undecidable for such languages.

Considerable effort has been spent by the programming languages community [2, 3, 5, 7] to develop systems which attempt to mitigate this problem for Python. These solutions rely on formal logic to prove type assertions for variables throughout the program. In many cases, such techniques can effectively reason about types statically. However, such approaches are necessarily conservative, and usually make certain restrictive assumptions about the program.

MINO takes a radically different approach to the problem of static type inference for dynamic languages, specifically Python. Instead of relying on formal semantics, MINO takes a data-driven approach based on machine learning techniques. The main intuition is that there are many aspects of a program which provide information about types, beyond formal semantics. For instance, properties like variable names and certain usage patterns all collectively provide hints as to a variable's type. As a concrete example, variables named `i` are usually index variables or counters and thus integers. While this kind of data-driven approach is only an approximation, we believe such a system has several potential strong use cases.

Our envisioned use case is for optimizing the generation of native machine code. For instance, many modern dynamic language interpreters implement a form of just-in-time (JIT) compilation, where instead of interpreting programs solely based on abstract syntax trees (ASTs) or high level bytecode, certain fragments of the program are compiled to equivalent native machine code. These fragments are often generated with a *fast-path* case where a variable is assumed to hold a certain type, and a *slow-path* case which serves as a fallback if the fast-path assumption does not hold. Thus, predicting a type incorrectly here does not sacrifice correctness, but rather incurs a performance penalty. Therefore, as long as MINO's type predictions are mostly correct, then we believe that such a system can provide performance gains for native code generation, beyond what a traditional type inference system can. This is the advantage of not being overly conservative with predictions.

The remainder of this paper will focus on the design and implementation of MINO, which to the best of our knowledge is the first system which uses machine learning to perform type inference for programs written in a dynamically typed language. We present an evaluation which shows that MINO is able to achieve reasonable predictive performance on a wide variety of different programs.

2 RELATED WORK

To the best of our knowledge, MINO is the first system which attempts to use machine learning techniques to perform static type inference for Python programs.

There is, however, a wide body of work on using traditional programming language techniques to perform static type inference for Python programs. In this section, we outline a few of the approaches:

Cannon [3] presents modifications to the cPython interpreter, which check for statements that guarantee a variable take on a certain type, such as equality checks and assignment to literals. Cannon's algorithm is based on propagating these type guarantees across control flow boundaries. This approach is necessarily conservative, however, because unless the system sees direct assignments or other equality-like checks to a variable, it is unable to make any specific type assertions about that variable.

Aycock [2] proposes type inference for a restricted subset of Python. Specifically, Aycock relies on flow insensitivity and type consistency within a scope. These two rules together disallow programs to change types via control flow. The drawback of this approach is that it requires restricting the functionality of Python to make type inference more tractable.

Rees [7] takes advantage of the pre-defined interface of builtin Python functions, and builds a type system around these interfaces. For instance, a variable passed to builtin `len()` must have a `__len__()` method which takes zero arguments and returns an integer. Type inference works by gathering as many facts about variables as is possible statically, and applying Rees's type system rules to reach type conclusions.

Maia [5], like [2], performs type inference on a restricted subset of Python called RPython, which is the restricted language which the PyPy [1] interpreter is written in. RPython, while being quite a powerful subset, still has a number of limitations, such as not allowing nested functions or any operator overloading.

It is important to note that the systems described above are all complementary to MINO. Combining both traditional type inference techniques with MINO's techniques is an interesting direction for future work, and in §3.3 we describe one place in MINO where this combination could take place.

3 DESIGN

This section discusses the high level design of MINO. MINO works by using machine learning techniques to train type classifiers based on pre-existing Python programs, which are given to MINO as input during the training phase.

To make the system feasible to implement, we made a few simplifying assumptions, which we outline below.

3.1 Simplifying assumptions

Restricting the possible types. In MINO, we restrict the possible types we can classify to the builtin primitive types (`int`, `str`, `list`, `dict`, etc.) and `object`. We treat `int` and `long` as the same (integer) type, and we also treat `str` and `unicode` as the same (string) type. Finally, we treat user-defined objects as belonging to the type `object`. There are several reasons for this restriction. First, we wanted a finite number of types (there are an infinite number of user-defined object types), so that training a classifier would be feasible. Second, we wanted MINO to extend to new programs which MINO's classifiers were not trained on.

We believe that predicting specific object types can be solved in the following two-level approach: First, run MINO to determine

whether or not a variable belongs to the base `object` type. Second, if it does, run a separate algorithm to determine which specific object type is most likely. We believe this second algorithm is much simpler to design, because a straightforward scheme such as matching method calls using whole program analysis would most likely perform well. Furthermore, the benefits to predicting user types is less clear for our envisioned use case; specialized native instructions exist for primitive types such as integers and strings, but not so much for custom user types.

Program correctness. We assume that input programs are well-typed. While MINO can still take as input a program which has type errors, it will most likely not perform as well.

3.2 Labelled data and feature generation

MINO relies on analyzing pre-existing Python programs to train its type classifiers. Here we describe how we take an existing Python program and generate labelled data points for training. Our main approach for generating labelled data points is to run the training programs and collect type information as the program executes. Before executing the training programs, we re-write each program's AST to wrap every load of a variable with an instrumentation function, which records the variable's name, location in source code, and its runtime type. While this imposes a performance overhead during execution, this is acceptable since training is an infrequently performed task.

This information alone, however, is insufficient. We need to be able to map each occurrence of a variable to the scope it belongs to. This is because MINO keeps track of features on a per-scope level. To deal with this, we uniquely number each scope within a file and keep track of the mapping between source code location and the scope which the symbol is defined in. This allows us to generate features on a per-scope level. This is best illustrated with the example show in Figure 1.

```
1 def fib(n):
2     if n == 1 or n == 2:
3         return 1
4     return fib(n - 1) + fib(n - 2)
```

Figure 1: `fib.py`

For `fib.py`, MINO records the following mapping information, assuming the scope of `fib()` has been assigned a scope id of 0:

$$(\text{var=n, file=fib.py, line=2, col=7}) \rightarrow (\text{file=fib.py, scope=0})$$
$$(\text{var=n, file=fib.py, line=2, col=17}) \rightarrow (\text{file=fib.py, scope=0})$$
$$(\text{var=n, file=fib.py, line=4, col=15}) \rightarrow (\text{file=fib.py, scope=0})$$
$$(\text{var=n, file=fib.py, line=4, col=28}) \rightarrow (\text{file=fib.py, scope=0})$$

The decision to keep track of features on a per-scope basis is a compromise. On the one hand, per-scope tracking allows us to record more observations for a particular variable. However, because Python scopes are, unlike C/C++/Java, per function instead of per block, it causes us to perform poorly in situations like those shown in Figure 2. In Figure 2, all occurrences of `x` belong to the same scope, but the type of `x` is dependent on which branch is taken. This kind of programming is mostly a consequence of developers used to C-style lexical scoping, which would place each instance of `x` within its own private scope. It is worth noting, however, that most, if not all, of the systems described in §2 perform poorly on or disallow `switchtype.py`.

One additional complication with this approach has to deal with object attributes, i.e. what scope do we tie `x` in `self.x` to? We deal with this problem by associating `x` with `self` in `self`'s scope. This does not allow us to handle chains with method invocations, however. For instance, in the expression `self.x.y().z`, we would be able to bind variables `self`, `x`, and `y` to a scope, but not variable `z`.

```
1 def switchtype(t):
2     if t == "int":
3         x = 123
4     elif t == "string":
5         x = "hello"
6     else:
7         x = [1, 2, 3]
8     return x
```

Figure 2: `switchtype.py`

3.3 Feature selection

We now turn to the question of what features to assign to each symbol in a scope. Since our goal is to build a system which works at compile time, we can only consider features based on the AST and the results of any static program analysis. Note that the features we used are mostly binary, with a few categorical features. We do not use any real valued features.

Features based on symbol name. We explore a variety of different features based on the name of the variable only. Some of the features we considered were detecting the use of Hungarian notation [1] and extracting type based on the prefix, whether or not the name was in all capital letters, whether or not the name began with a set of common prefixes (`is_`, `has_`, etc.), and whether or not the name matched a set of common variable names (`i`, `n`, `f`, `self`, etc.) Ultimately we decided on only using the last feature, since we found very little usage of Hungarian notation in the programs we trained on, and the common prefixes were usually used for member functions.

Features based on AST. Most of the interesting features unsurprisingly come from this category. We consider all of the AST nodes in Python, and during an AST traversal we emit the corresponding features. For example, for the following expression `x[i]`, we emit a feature called *is_subscriptable* for variable `x`, and a feature called *is_subscript_index* for variable `i`. Figure 3 shows a summary of the different kinds of features we consider. Below, we discuss a few of the more interesting feature selection choices we made.

Expression	Features generated
x op y	**x**: used_in_*op*, **y**: used_in_*op*
x[i]	**x**: is_subscriptable, **i**: is_subscript_index
x()	**x**: is_callable
while x:, if x:	**x**: used_in_truth_test
for e in elems:	**e**: is_loop_target, **elems**: is_loop_iter
x in y:	**x**: is_search_key, **y**: is_searchable
x.y:	**x**: used_as_object
x, y = z	**z**: used_as_multiassign_rhs
x = literal	**x**: assigned_*type*_literal
print x	**x**: used_in_print
return x	**x**: is_return_value
yield x	**x**: is_yield_value
{ x : y }	**x**: used_as_dict_key
def f(x):	**x**: is_func_arg0
builtin_function(x):	**x**: used_as_builtin_*func*_arg0

Figure 3: Summary of features generated by MINO

[1] https://en.wikipedia.org/wiki/Hungarian_notation

Binary operations turned out to generate the most interesting features, so we specialize the features further depending on constants in the expression. For example, the expression `x == y` generates a feature *used_in_eq* for both variables `x` and `y`, whereas the expression `x == 1` generates a more specific feature *used_in_eq_int* for variable `x`. We use a similar set of features for the other binary operators, such as `<`, `+`, `**`, etc.

Like [7], we also take advantage of the pre-defined builtin functions such as `len()`, `range()`, `min()`, etc. For builtin functions, we emit a feature if a variable is used as an argument to a builtin. For example, for `len(x)`, we emit the feature *used_as_builtin_len_arg0* for `x`. Note that we are different from [7] in that we do not explicitly encode any type rules about the builtin functions into MINO. Instead, we learn these type rules from the training data.

Finally, we handle assignments specially. In the simple case, if we see `x = literal`, we emit a feature *assigned*_type_*literal*. However, certain functions are known to return specific types. For instance, `hash()` and `id()` always return integer types, and `open()` always returns a file object. Thus, we treat `x = 10` the same as `x = hash(o)`; both are equivalent to assigning `x` to an int literal. This is the one part of MINO where we do specifically encode knowledge of the builtin API.

MINO does a very simple form of assignment type propagation. We keep track, per symbol, of any literal type assignments. Any subsequent expressions which reference the variable can lookup its assigned literal type, in order to generate more specialized type features. For example, in the statement `x = 10; y = x`, we would generate *assigned_int_literal* features for both `x` and `y`. We do not implement any constant folding of type assignments, although doing so would be straightforward. More generally, we do not propagate features among symbols. That is, if we see `x = y`, we could in principle copy, share, or merge the features between `x` and `y`. Exploring these kinds of ideas is an example of combining MINO's approach to type inference with the more traditional programming language techniques mentioned in §2, and is an interesting direction for future work.

A simple example. To illustrate these ideas, consider the Python snippet shown in Figure 4.

```
1  def argmax(elems):
2      """Pre-condition: elems is not empty"""
3      min_pos, min_elem = 0, elems[0]
4      for i in range(1, len(elems)):
5          if elems[i] < min_elem:
6              min_pos, min_elem = i, elems[i]
7      return min_pos
```

Figure 4: `argmax.py`

For the `argmax.py` snippet, the following features are generated (assuming `argmax` is assigned scope id 0). Note we abbreviated the feature names for conciseness:

(var=elems, scope=0) : [farg0, subscript_int, subscript, len_arg0]
(var=min_pos, scope=0) : [assign_int, return_value]
(var=min_elem, scope=0) : [binop_compare]
(var=i, scope=0) : [name_is_i, loop_target, subscript_index]

The features generated from Figure 4 illustrate another point: we currently do not merge features together. That is, for the variable `elems`, *subscript_int* is a stronger feature than *subscript*, but we currently keep both features around. Whether or not discarding weaker features would be helpful is something that we have not explored.

3.4 Learning type classifiers

§3.3 outlines how MINO generates labelled data from Python programs. In this section we explore how MINO transforms labelled data into classifiers for type inference. For this section, let $X = [x^{(1)}, ..., x^{(d)}]$ represent the labelled data feature vectors, and let $Y = [y^{(1)}, ..., y^{(d)}]$ represent the labelled classes.

Naïve Bayes (NB) classifier. We start with a classic Naïve Bayes classifier, because we believe that for this problem domain, the NB assumption of feature independence given a class is quite reasonable. Given the type of a variable, the probability that a certain feature is present is, intuitively, mostly independent of the other features present. For instance, if an integer variable is used in bit-shifts, that does not really give more or less evidence as to whether or not that variable will be used as a list index. Of course, there are cases where this is not true; observing an integer variable in a bit-shift might give more probability for observing it in an addition, for example.

Recall that the NB decision function is given by:

$$\arg\max_y \log P(y) + \sum_{i=1}^{n} \log P(x_i|y)$$

This is another reason why NB is appealing for this problem, because it is inherently a multi-class classifier. To train a NB classifier, we learn the class priors $P(y)$ and feature conditional probabilities $P(x_i|y)$ by doing frequency counting on the training data, using some form of α-smoothing.

Support Vector Machine (SVM) classifier. In addition to a NB classifier, we also explored using a multi-class SVM classifier. Recall that the standard SVM formulation is a binary classifier, with a decision function given by:

$$\text{sign}\left[\sum_{i \in SV} y_i \alpha_i K(x_i, x) + w_0\right]$$

There are various strategies for extending a binary SVM to do multi-class classification, including *one-vs-one* [4] and *one-vs-all* (OVA) [9]. We use the strategy of OVA, which trains a binary SVM for each class, and picks the class corresponding to the binary SVM which assigned the greatest margin for the query point. OVA is preferable because it allows us to train less classifiers (exactly $|y|$) than the one-vs-one strategy ($O(|y|^2)$ classifiers). We use the standard procedure of k-fold cross-validation to pick the appropriate hyper-parameters.

To deal with class imbalance, we use the strategy found in [6] which scales the regularization term C for a class's binary SVM to be inversely proportional to the frequency of the class in the training data. Intuitively, this penalizes the more frequently occurring classes by increasing the regularization for these classes relative to the less frequent classes.

Other options explored. NB and SVM were the two classifiers which gave us the best results. We also tried a wide variety of different parametric learning strategies. We did not explore non-parametric approaches, due to the high dimensionality of the feature vectors (we believe that k-nearest-neighbor and decision tree approaches would perform very poorly on our dataset).

One interesting approach we tried, which turned out to have poor performance, was an approach based on 1-Class-SVMs [8]. The idea was to treat the problem as a outlier detection problem. For each class, we trained a 1-Class-SVM using only the labelled points

for that class. A prediction was done by querying each class's 1-Class-SVM, and assigning a label to the class which indicates that the query point was not an outlier. We broke ties arbitrarily, though in principle one could look at distance from the separating hyperplane. We believe this approach did not perform well because there were too few data points with respect to the feature space dimensionality for some classes, and thus their 1-Class-SVMs were not very useful.

We also tried to use linear discriminant analysis (LDA), which, like NB, is appealing because it is inherently multi-class. However, LDA also performed quite poorly. We believe this is because the Gaussian assumption for the distribution of $P(x_i|y)$ is a very poor assumption for binary/categorical features. NB avoids this issue because it constructs a binomial/multinomial distribution for $P(x_i|y)$.

4 IMPLEMENTATION

MINO is implemented in two separate modules.

The first module is responsible for extracting the information described in §3 out of *unmodified* Python programs. We do this by modifying PyPy [1], the popular open-source Python interpreter (written in Python itself). We chose to build on top of PyPy because it is simpler to modify and more extensible than cPython, the official Python implementation. We implement feature extraction and variable instrumentation as a series of AST traversals and transformations within the PyPy interpreter. This allows us to take advantage of PyPy's semantic analysis functionality, which gives us access to symbol tables and scope information. Overall, we added a modest $\sim 1,000$ lines of code to PyPy.

The second module is responsible for taking the labelled data generated by the first module, and training a set of type classifiers. To do this, we use scikit-learn [6], a powerful open-source Python library for machine learning.

We plan to release both modules as open-source software.

5 EVALUATION

This section presents a thorough evaluation of MINO on a collection of open-source Python projects. Here, we evaluate the success of the various classifiers discussed in §3.4.

5.1 Training and validation projects

We made a best-effort attempt to collect a non-biased sample of open-source Python projects. That is, we tried to vary the domain of projects so as to not introduce any specific biases (authors of projects in a particular domain might have specific coding practices, or a particular domain might be biased towards certain types of operations). Unfortunately, we could not include any GUI based programs, because we were unable to get PyPy's Tkinter support to work properly.

Figure 5 and Figure 6 show a summary of the open-source projects we used for training and validation, respectively. For each project, we instrumented its code base and ran the project's test suite. We had 20 projects in total, and we trained on 15 of the projects.

Project	Version	Short Description	LOC
flask	0.9	Web framework	5,581
nzmath	1.2	Number theory library	22,363
rocket	1.2.4	Threaded web server	2,954
twisted	12.2.0	Event-driven library	156,160
aima-python	-	Algorithms from AIMA	3,501
beautifulsoup4	4.1.3	HTML parser	4,405
hsautotag	1.1.1	Metadata extraction	1,485
httplib2	0.7.7	HTTP client library	5,010
mpmath	0.17	Multiprecision library	29,098
pyDatalog	0.11.2	Logic programming	2,310
spambayes	1.1a6	Spam filtering	46,715
sympy	0.7.2	Symbolic math	186,809
tfidf	1.1	TF-IDF	135
tornado	2.4.1	Non-blocking web server	12,866
wheezy.template	0.1	Templating library	1,816

Figure 5: Projects used by MINO for training

Project	Version	Short Description	LOC
biopython	1.60	Comp-bio libraries	117,003
pychecker	0.8.18	Python linter	9,680
pyparsing	1.5.6	Parser library	11,093
python-markdown	2.2.1	Port of Markdown	10,032
pytz	2012h	Timezone library	2,491

Figure 6: Projects used by MINO for validation

5.2 Experimental methodology

Our experimental methodology is as follows. We evaluate both the NB classifier and the SVM classifier described in §3.4. For each classifier, we run $k = 5$ cross-fold validation in a grid search to find the best hyper-parameter values. We use the F1 score metric, which combines both precision and recall, as the scoring metric to optimize. Within the training data, we also set aside 25% of the data points for validation. Thus, §5.4 shows results for two different validation sets: one set of points which is sampled from the same projects which the training data comes from, and another set of points which is sample from never seen before projects. We will call the former $V1$, and the latter $V2$.

For the NB classifier, the only hyper-parameter is the smoothing parameter α. Here, our grid search consists of $\alpha = [0.0001, 0.001, 0.01, 0.1, 1.0]$.

The SVM classifier has, unfortunately, quite a few hyper-parameters. There is the C hyper-parameter, which controls the hardness of the SVM. Additionally, there are several different standard kernels to choose from, each with a different set of parameters. In the interest of simplicity, we only consider a linear kernel with a grid search of $C = [10^{-1}, 1, 10, 100, 1000]$.

We also compare our classifiers to a baseline classifier which always predicts the most frequently occurring class. This is interesting because there is a class imbalance skew in favor of type `object` (see Figure 8).

Finally, we do not consider, in both training and validation, data points which have no feature values. For instance, consider the snippet shown in Figure 7. In this snippet, `myvar` has no features, since its only observation in the code is being stored with the return value of an unknown type or function. If a variable has no features, then MINO will not be able to make any reasonable decisions. These cases are not common, however, since variables are usually created to be used. For instance, in our validation dataset, this occurred roughly $\sim 5\%$ of the time.

```
1 def foo():
2     myvar = bar()
```

Figure 7: `nofeatures.py`

5.3 Data preprocessing

Collecting data points by running the training programs introduces a slight complication with repeated points. Loops and multiple invocations of the same function (very common with library code) can cause the same data point to be generated repeatedly. Because we do not want this to bias our training data, we only allow each data point to appear at most once per source code location.

Furthermore, because our training projects vary in complexity and code coverage of test suites, a simple merging of all labelled data from the training projects would heavily bias the data points towards a few particular projects. To control for this, we do not allow any particular project to contribute more than $5,000$ data points to the training data. If a project has more than $5,000$ data points, we uniformly sample from it. Otherwise, we take all the data points from the project. $5,000$ was picked empirically, since it is a reasonable number that most projects can achieve. For the validation data, we simply include all the data points.

After pre-processing, our training dataset had $\sim 20,000$ labelled data points.

5.4 Results

In this section, we report the precision, recall, and F1 scores for the best hyper-parameter values for both the NB and SVM classifier, on both of the validation datasets ($V1$ and $V2$). Recall that F1 is the harmonic mean of precision and recall. Because of the class imbalance (see Figure 8), we do not report accuracy scores.

We also show the confusion matrix for each classifier on the $V2$ validation dataset. Recall that for a confusion matrix C, C_{ij} is the number of points belonging to class i, but predicted to belong to class j. The ideal confusion matrix is a diagonal matrix. Rows in confusion matrices represent the recall of a class, whereas columns represent the precision of a class.

5.4.1 Baseline

Figure 8 shows the prior distribution of class types from the training data.

Type	Frequency	Type	Frequency
object	.656	tuple	.025
int	.099	bool	.017
str	.097	float	.008
list	.050	set	.004
dict	.042	file	.001

Figure 8: Prior distribution of types from training data

Given the prior distribution, the baseline classifier always predicts type `object`. Figure 9 shows the classification scores for the baseline classifier. We omit the breakdown of results by types here and only report an aggregate, since this classifier only predicts one class. Here we see that the baseline classifier only achieves a F1 score of roughly 0.5 on the validation data.

Dataset	Precision	Recall	F1
$V1$	0.44	0.67	0.53
$V2$	0.40	0.64	0.49

Figure 9: Scores for the baseline classifier

5.4.2 NB results

For the NB classifier, the optimal grid search value was at $\alpha = 0.001$. Figure 10 shows the classification scores for the NB classifier. Figure 11 shows the confusion matrix. Here, we see very reasonable F1 scores for both the $V1$ and $V2$ datasets.

5.4.3 SVM results

For the SVM classifier, the optimal grid search value was at $C = 10$. Figure 12 shows the classification scores for the SVM classifier, and Figure 13 shows the confusion matrix. Comparing F1 scores with the NB classifier, we see that the SVM classifier just slightly out-performs. We believe these scores between NB and SVM are similar enough that the differences are just noise; NB and SVM are both reasonable classifiers to use in MINO.

Dataset	Type	Precision	Recall	F1	Support
$V1$	bool	0.52	0.85	0.65	75
	dict	0.74	0.83	0.78	190
	file	0.40	0.33	0.36	6
	float	0.36	0.36	0.36	50
	int	0.72	0.80	0.76	419
	list	0.74	0.75	0.74	238
	object	0.98	0.88	0.92	3112
	set	0.36	0.25	0.30	16
	str	0.53	0.84	0.65	417
	tuple	0.54	0.45	0.49	143
	-	**0.86**	**0.83**	**0.84**	4666
$V2$	bool	0.55	0.94	0.70	279
	dict	0.70	0.82	0.75	682
	file	0.55	0.28	0.37	61
	float	0.76	0.30	0.43	241
	int	0.56	0.70	0.62	1111
	list	0.76	0.75	0.76	1221
	object	0.97	0.86	0.91	11208
	set	0.38	0.74	0.50	19
	str	0.61	0.79	0.69	2587
	tuple	0.23	0.43	0.30	206
	-	**0.85**	**0.81**	**0.82**	17615

Figure 10: Scores for the NB classifier

	bool	dict	file	float	int	list	object	set	str	tuple
b	263	0	0	0	4	0	1	0	11	0
d	1	556	0	0	0	32	42	2	34	15
f	1	0	17	0	0	2	33	0	8	0
flt	1	0	0	73	142	0	2	0	20	3
int	43	0	0	18	773	0	11	0	222	44
l	27	84	0	1	14	921	19	10	109	36
obj	67	118	11	3	329	122	9592	6	873	87
set	1	1	0	0	0	2	1	14	0	0
str	68	21	3	1	117	70	145	4	2048	110
tup	2	12	0	0	6	68	0	1	28	89

Figure 11: Confusion matrix for the NB classifier

5.4.4 Analysis

To provide some intuition for which features are the most predictive for type inference for each class, we show the top three features for each class that the NB classifier has assigned the highest conditional probability $P(x_i|y)$ to. Figure 14 summarizes the results, showing a representative expression for variable `x` which results in the feature being emitted.

From Figure 14, we immediate see why MINO has trouble distinguishing say, integers from floats, or tuples from lists. This is because the most likely features are sometimes shared amongst different types. For example, both integers and floats are often (not surprisingly) used in multiplication and addition. This is complicated by the fact that Python allows for implicit type promotion, so integers and floats are often mixed in arithmetic expressions.

To better understand the predictive capabilities of MINO amongst different classes, it helps to look at confusion matrices. Figure 11 and Figure 13 are the confusion matrices for the NB and SVM classifiers, respectively. In analyzing the confusion matrices, we gain a better understand of exactly which types are hard to distinguish from other types.

For instance, consider the `str` type. We see, by looking at the `str` columns in the confusion matrices, that it suffers from precision issues when it predicts type string when the actual type is either integer or list. This makes sense in the context of Python. Both integers and strings support the binary addition operator, and both strings and lists can be use very similarly (i.e. iteration with for loops, passed to `len()`, subscripted and sliced).

Dataset	Type	Precision	Recall	F1	Support
$V1$	bool	0.38	0.87	0.53	75
	dict	0.70	0.84	0.76	190
	file	0.12	0.33	0.17	6
	float	0.20	0.48	0.28	50
	int	0.77	0.71	0.74	419
	list	0.76	0.70	0.73	238
	object	0.99	0.87	0.93	3112
	set	0.30	0.50	0.37	16
	str	0.55	0.79	0.65	417
	tuple	0.50	0.50	0.50	143
	-	**0.87**	**0.82**	**0.84**	4666
$V2$	bool	0.33	0.95	0.49	279
	dict	0.68	0.83	0.75	682
	file	0.33	0.26	0.29	61
	float	0.28	0.52	0.37	241
	int	0.63	0.60	0.62	1111
	list	0.76	0.73	0.75	1221
	object	0.99	0.86	0.92	11208
	set	0.17	0.79	0.28	19
	str	0.67	0.79	0.72	2587
	tuple	0.25	0.49	0.33	206
	-	**0.86**	**0.81**	**0.83**	17615

Figure 12: Scores for the SVM classifier

	bool	dict	file	float	int	list	object	set	str	tuple
b	264	0	1	0	3	1	1	0	8	1
d	3	563	1	0	0	34	15	11	41	14
f	1	0	16	0	0	5	23	0	16	0
flt	3	6	2	125	71	0	5	0	21	8
int	58	2	5	130	669	1	5	0	191	50
l	28	94	2	13	5	892	10	17	110	50
obj	302	117	14	118	204	117	9613	34	614	75
set	1	1	0	0	0	2	0	15	0	0
str	127	24	8	49	107	67	51	12	2035	107
tup	4	15	0	5	1	53	3	1	23	101

Figure 13: Confusion matrix for the SVM classifier

Another example comes from the `tuple` type. By looking at the `tuple` row in the confusion matrices, it is clear that tuple types have recall issues particularly with list types. Once again, this is not surprising, since tuples and lists can be used almost interchangeably, with the only major difference between that tuples are immutable. Thus, only when MINO sees a list/tuple-like variable being mutated, can it detect that the variable is a list instead of a tuple. In the absence of such mutation, MINO prefers to predict list, because list is more commonly used than tuple (see Figure 8).

6 Future Work

There are a lot of possible areas of future work for MINO-like systems. The first set of improvements involve further fine-tuning of the classifiers via a combination of better feature selection and feature space reduction. Being able to distinguish between similar types such as lists and tuples would greatly improve the predictive power of MINO.

Type	1st ranking	2nd ranking	3rd ranking
bool	if x:	x = True	x or y
dict	x["hello"]	x.get(k)	x[k]
file	x = open(...)	x.close()	variable named f
float	x * y	x == 1	x + y
int	x * y	x « y	x + y
list	x[10]	x.append(e)	x = []
object	x.attr	x.meth()	variable named self
set	x.add(e)	for e in x:	x = set()
str	x.join(...)	def f(y, x):	for c in x:
tuple	x == y	x[10]	x = (...)

Figure 14: Features with the highest class conditional probabilities

As mentioned previously in §3.3, there is also another direction of improvements which try to incorporate more traditional program analysis techniques during feature generation. MINO is designed to be mostly agnostic to the language semantics. For instance, nowhere do we encode that assignment of a variable to a literal value allows us to immediately deduce the type of the variable; we only emit a literal assignment feature, and the correlation with the actual variable type is learned during training. Whether or not this kind of agnostic approach is the best approach to this problem is an open question.

Another possibility of future work involves integrating MINO into a Python JIT compiler, such as PyPy. It remains to be seen whether or not MINO's predictive power is sufficient enough for JIT compilers to actually benefit from. For example, it may be the case that the time spent in generating type predictions offsets any potential performance gains. This is yet another reason we limited our classifiers to simple classifiers. Exploring the trade-offs between a more powerful classifier and the computational resources required is an interesting problem.

7 Conclusion

This paper presents MINO, a new system for performing static type inference on Python programs. MINO approaches the problem of type inference by treating type inference as a multi-class classification problem, and applies machine learning techniques to construct classifiers trained on a wide variety of open-source Python projects. We demonstrate that such an approach is practical and performs well, achieving a F1 score around ~ 0.8 on various Python programs.

References

[1] Pypy. URL `http://pypy.org/`.

[2] J. Aycock. Aggressive type inference, 2001. URL `http://www.python.org/workshops/2000-01/proceedings/papers/aycock/aycock.html`.

[3] B. Cannon. Localized type inference of atomic types in python. Master's thesis, 2005.

[4] S. Knerr, L. Personnaz, and G. Dreyfus. Single-layer learning revisited: A stepwise procedure for building and training a neural network. In *Neurocomputing: Algorithms, Architectures and Applications*, 1990.

[5] E. Maia, N. Moreira, and R. Reis. A static type inference for python. In *Proc. of DYLA*, 2012.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[7] G. Rees. Type inference for python, 2002. URL `http://garethrees.org/2002/02/26/type-inference`.

[8] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the support of a high-dimensional distribution. *Neural Comput.*, 13(7):1443–1471, July 2001. ISSN 0899-7667. doi: 10.1162/089976601750264965. URL `http://dx.doi.org/10.1162/089976601750264965`.

[9] V. Vapnik. *The Nature of Statistical Learning Theory*. 1995.