# SATD

### [An Exploratory Study on Self-Admitted Technical Debt](An%20Exploratory%20Study%20on%20Self-Admitted%20Technical%20Debt.pdf)

Aniket Potdar, Emad Shihab, 2014

Paper that introduces SATD.

They find how often SATD occurs and that it is created by multiple developers, often created by experienced developers, throughout the whole project. They mention 62 different patterns of text that indicate SATD. Complexity of the project does not seem to correlate with SATD.

In the discussion mention 4 scenarios of inconsistent SATD updates:

TD-Code standing for the incorrect code.

1. SATD is removed with the TD- Code
2. SATD is removed but code is unchanged
3. SATD unchanged while TD-Code is removed
4. SATD unchanged and code unchanged

### [Was Self-Admitted Technical Debt Removal a real Removal? An In-Depth Perspective](Technical%20Dept%20Removal.pdf)

Fiorella Zampetti, Alexander Serebrenik, Massimilani Di Penta, 2018

About how “self-admitted” technical debt has been removed.

It shows that for the majority, removal of the SATD occurs with change of the code, often however the whole class or method is removed. But still for more than 30% (and often the majority), the method is changed. This means that the classifying problem is a valid one.

Most removals of SATD’s is not mentioned in commit messages. (only 8% is)

The types in which source code is altered can belong to 6 categories, about 50% of them are in the “other” category.

1. Changing method calls
2. Conditional statements: Addition, removal or change to preconditions
3. Method Signatures: Either the parameters or the return type of the SATD-affected methods is changed, for example because of a refactoring action or another improvement.
4. Try-Catch blocks: Addition of exception handling for previously-uncaught exceptions
5. Return statements: the return object is changed to improve the method functionality
6. More complex change / other

For the two first categories, the categories can be split into even more subcategories.

### [Automatically Learning Patterns for Self-Admitted Technical Debt Removal](Technical%20Debt%20Deep%20Learning.pdf)

Fiorella Zampetti, Alexander Serebrenik, Massimilani Di Penta, 2019

SATD removal is a necessity and often done by a developer that did not introduce it, but it often follows specific patterns. This study aims to categorize the categories.

A CNN trained on the embeddings extracted from the SATD comments and an RNN trained on embeddings extracted from the source code.

Data: 887 manually-classified method levels SATD removals

**CNN:**

For the word embeddings a skip gram model is used, then a CNN is trained using the embeddings.

*No pre-training*

The CNN achieves a precision of 39% and recall of 41% and an AUC of 0.61

**RNN:**

The tokens that are used for the word embeddings are chosen such that frequent terms are used more often. There are other strategies that build sparse embeddings or lose too much information

Then a skip gram model is used on the tokens to extract the embeddings for the source code.

Afterwards a RNN is trained on these embeddings.

The RNN achieves a precision of 47% and recall of 44% and an AUC of 0.64

Together the models achieve a precision of 55% and recall of 57% and an AUC of 0.73 on average with as high as up to 73% precision, 63% recall and 0.74 AUC

The CNN model performs better than manual classification of the models, both models together also perform better than manual classification.

# Code Embeddings

## Word2Vec

### *[Semantic Source Code Models Using Identifier Embeddings](Semantic%20Source%20Code%20Models%20Using%20Identifier%20Embeddings.pdf)*

Vasiliki Efstathiou and Diomidis Spinelli, Apr 2019

Applying Word2vec’s Skipgram to code, building embeddings of code. Method uses tokenization of the code and doesn’t use any of the structural information of code. C-BOW indirectly uses bag of words assumption, this might not hold for code. Dimensionality of vectors is 100, character n-gram is between 3 and 6. The window is 5 for all languages but python, where it is 4.

### [*SCOR Source Code Retrieval With Semantics and Order*](SCOR%20Source%20Code%20Retrieval%20With%20Semantics%20and%20Order.pdf)

Shayan A. Akbar & Avinash C. Kak 2019

Using Markov Random Fields and Word2Vec combined. It checks the ordering of terms in a query by using the Markov Random Fields approach.

This is used for Bug Localization, so the problem here is also in the Information Retrieval part. Finding the bugs in the first place.

## Path-Based Representations

[Code2Vec](#_code2vec_Learning_Distributed_2)

[Code2Seq](#_code2vec_Learning_Distributed)

### *[PathMiner A Library for Mining of Path-Based Representations of Code](PathMiner%20A%20Library%20for%20Mining%20of%20Path-Based%20Representations%20of%20Code.pdf)*

Vladimir Kovalenko et al. 2019, May

A library that makes it easy to mine the path-based representation of code. This tool can be used to simplify making paths, works on Java and Python.

## MLP

### [code2vec Learning Distributed Representations of Code](code2vec%20Learning%20Distributed%20Representations%20of%20Code.pdf)

Alon et al 2018

Learning code embeddings from code with (input X: the path representation of code) and (Y: the embedding of the name of the method). Using a path-based soft attention mechanism is the key in this paper.

## Encoder Decoder

### [Code2Seq: Generating Sequences From Structured Representations of Code](CODE2SEQ%20Generating%20Sequences%20From%20Structured%20Representations%20of%20Code.pdf)

Alon et al 2019

Encoder Decoder model with code not as tokens, but as AST’s represented as vectors. The model is expecting a tree structure. It translates to a natural language representation of what happens in the code. It argues that it can generate unseen sequences better than the previous approach in [code2vec](#_code2vec_Learning_Distributed). Possibly it also has a better embedding for code.

It’s a typical Encoder decoder, with most likely an RNN used.

### [An Empirical Study on Learning Bug-Fixing Patches in the wild via Neural Machine Translation](An%20Empirical%20Study%20on%20Learning%20Bug-Fixing%20Patches.pdf)

Tufano et al. 2018

An encoder-decoder model is trained to translate buggy code into fixed code. The code is formatted as AST edit operations, this is different than the AST in [code2vec](#_code2vec_Learning_Distributed_1) and [code2seq](#_Code2Seq:_Generating_Sequences). The model can generate patches for unseen code. With more than 82% of the produced code being correct. This method can be used to create run-able code. The model architecture uses RNN’s for the Encoder and the Decoder parts.

## Siamese Networks

### [Learning Semantic Vector Representations of Source Code via a Siamese Neural Network](Learning%20Semantic%20Vector%20Representations%20of%20Source%20Code%20via%20a%20Siamese%20Neural%20Network.pdf)

David Wehr, Halley Fede, Eleanor Pence, Bo Zhang, Guilherme Ferreira, John Walczyk and Joseph Hughes, Apr 2019

In this paper they use a Siamese network to train a model to make predictions on.

Firstly, they get the AST from the code, they flatten it into sequences using a method called “Structure-based traversal” so it can feed into the RNN.

Then the model is pretrained on large unlabeled datasets using an autoencoder network to get the initial weights for the model.

Afterwards the training is performed on the labelled data, where a Siamese network (shared weights) is used, the decoder part is removed, and a cosine similarity is calculated on the embeddings. As usual the training will separate the different classes in the embedding space.

The embedding layer is unclear but is mentioned to be a 128-dimensional LSTM. They achieve high AUC scores over 26 classes.

# Unread

### [A Convolutional Attention Network for Extreme Summarization of Source Code](A%20Convolutional%20Attention%20Network%20for%20Extreme%20Summarization%20of%20Source%20Code.pdf)

### [An Empirical Study On the Removal of Self-Admitted Technical Debt](An%20Empirical%20Study%20On%20the%20Removal%20of%20Self-Admitted%20Technical%20Debt.pdf)

S. Maldonado, R. Abdalkareem, E. Shihab and A. Serebrenik, 2017

# Not Useful

### *[Import2vec-Learning Embeddings for Software Libraries](Import2vec-Learning%20Embeddings%20for%20Software%20Libraries.pdf)*

Bart Theeten, Frederik Vandeputte, Tom van Cutsem, 2019

Builds word embeddings for programming libraries.

# Approach & Planning

## Definition

**Goal:** higher accuracy on classifying different classes of SATD. (See Question)

**Current statistics**: Average precision of 55% and recall of 57, AUC of 0.73

**Input**: combination of code and comments attached to this code. 887 SATD removals

**Output**: one of 6 classes

1. Changing API calls
2. Conditionals
3. Method Signatures
4. Exception handling
5. return statements
6. More complex change

**Limitations:** With just 887 data points, the current dataset is not that rich.

**Possible Solution:** Pretraining

**Possible Solution:** The word embedding trained from the SATD comments pre-trained on general comments in source code: Because it seems to me that the data set is too small to train a reasonable embedding. It is noted that no difference between using a pre-trained model and trained model is observed. This could also be due to a low quality of the embeddings. Perhaps we could train on more code-comment related data, and not just SATD comments.

**Possible Solution**: Build a good latent representation of code: Use a larger dataset to pretrain the model, perhaps autoencoders or variational autoencoders can be used to pretrain parts of the model. The choice in models to use for pretraining are interesting, perhaps code in general, perhaps bug fixes. For transfer learning to work, the task should be a similar task. An Autoencoder on code method reconstruction could be a starting point. One shortcoming of this method is that the latent variables for this task might not hold the same information to discriminate for the second task. E.g. the code information stored in the latent variable isn’t of any use of identifying the 6 different classes. It should be attempted and a compared to a baseline. (Option: Force the model to create compile-able code, can this compile-ability be backpropagated on? (Unit test) What does this Latent space mean?

**Possible Solution:** Not using Skipgram, but Path-based representation of Source Code: The source code can be encoded in different ways. The structure of source code is crucial. Research has shown that using path-based representations can work more effectively. This can be an approach, perhaps in combination with other methods.

**Possible Solution:** One Shot learning could be trained to the differentiation between the classes. If somehow in this representation the different classes can be of different distance to each other. Perhaps Triplets can be created, of anchor, similar and different to try to teach the model the differences within the classes to better differentiate between the classes.

## Questions

**Question:** The model could be trained to maximize results on the training set, if it would also be getting fed SATD’s during execution. If this is the case, then there must be another model that is taking care of SATD identification. This information is crucial for my next step, I could either get a model that is expecting to see SATD’s and only has to classify them. Did you already fix the issue of finding SATD’s? Or how will you address this? Perhaps I can think of a representation or method that can do both.

**Question:** How is the data fed into one neural network? A model with 2 inputs and one output?

**Question:** The dataset is imbalanced (>50% is other), what methods have you used to deal with this?

**Question:** SATD’s can belong to multiple classes, how is this dealt with in the model? Have you altered your SoftMax at the end of the architecture?

**Question:** Was the SATD comments data enough to train a word embedding? No difference between pre-training and no pre-training is observed, how was it pre-trained?

## Approach

1. Try out [Semantic Source Code Models](#_Semantic_Source_Code) pre-trained code embedding for the task
2. Using [code2vec](#_code2vec_Learning_Distributed_2) for the source code, instead of RNN and tokens
3. Using [Code2Seq:](#_code2vec_Learning_Distributed) for the source code to try to get latent set of features.
4. Using [BugFixes](#_An_Empirical_Study) AST representation for the decoder part.
   1. Using [PathMiner](#_PathMiner_A_Library) to easily get the path representations of the code and use this instead of tokenization.
5. Trying out [Siamese Network](#_Learning_Semantic_Vector)’s approach to help the embedding space to differentiate between the classes. If the other approaches don’t work.