Extracting Code from Programming Tutorial Videos

Shir Yadid
Extracting Code from Programming Tutorial Videos

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Shir Yadid

Submitted to the Senate of the Technion — Israel Institute of Technology
Shvat 5776 Haifa February 2016
The research thesis was done under the supervision of Prof. Eran Yahav, in the Computer Science Department.

The generous financial support of the Technion is gratefully acknowledged. The research leading to these results has received funding from the European Union’s - Seventh Framework Programme (FP7) under grant agreement n°615688 - ERC - COG - PRIME.
Contents

Abstract 1

Abbreviations and Notations 2

1 Introduction 3

2 Overview 6
   2.1 Extracting Raw Text 7
   2.2 Extracting Code from Raw Text Frames 8

3 Background 12
   3.1 Optical character recognition (OCR) 12
   3.2 Statistical language models 13
   3.3 Alignment 14

4 Model 15
   4.1 Likely Corrections of Programming Tutorial Videos 15

5 Our Approach 18
   5.1 Finding Possible Values of Token Variables 19
      5.1.1 Finding Possible Values for a Token 23
   5.2 Forcing Line Structure 24
   5.3 Fragment Level Correction 25

6 Implementation 27
   6.1 Android Framework 27
   6.2 Extracting Text Buffers from a Video 27
   6.3 Differing the Target Language 28
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4 Language Models: Training Phase</td>
<td>28</td>
</tr>
<tr>
<td>7 Evaluation</td>
<td>30</td>
</tr>
<tr>
<td>7.1 Methodology</td>
<td>30</td>
</tr>
<tr>
<td>7.2 Results</td>
<td>31</td>
</tr>
<tr>
<td>8 Related Work</td>
<td>37</td>
</tr>
<tr>
<td>9 Limitations</td>
<td>40</td>
</tr>
<tr>
<td>10 Conclusion</td>
<td>42</td>
</tr>
<tr>
<td>Hebrew Abstract</td>
<td></td>
</tr>
</tbody>
</table>
## List of Figures

2.1 The architecture of ACE. ........................................ 7
2.2 A frame extracted from example tutorial video. ............. 8
2.3 Noisy raw text buffers as extracted from video frames by applying OCR. ........................................ 9
2.4 Code Extracted by ACE. ........................................... 11

4.1 The symbolic version (where tokens are turned into variables) of the text buffer from Fig. 2.3(a). .......................... 16

7.1 Coverage and precision for varying threshold. ............... 34
7.2 Percentage of line structures considered valid as a function of number of occurrences in the model. ....................... 35
7.3 Top ten line structures. ........................................... 36
Abstract
Abstract

The number of programming tutorial videos on the web increases on a daily basis. Video hosting sites such as YouTube host millions of video lectures, with many programming tutorials for various languages and platforms. Despite the wealth of this information, and despite the fact that large parts of these videos contain code, there is no effective way to index and search such videos. There are two main challenges to effective indexing of programming tutorial videos: (i) code in tutorials is typically written on-the-fly, with only parts of the code visible at each frame, and (ii) optical character recognition (OCR) is not precise enough to produce quality results from videos.

We present a novel approach for extracting code from videos based on the following ideas: (i) consolidating code across frames, and (ii) a combination of statistical language models for applying corrections at different levels, allowing us to make corrections by choosing the most likely token, combination of tokens that forms a likely line structure, and combination of lines that lead to a likely code fragment in the language. We have implemented our approach in a tool called ace, and used it to extract code from 40 Android video tutorials on YouTube. A video tutorial on how ace works is available at https://vimeo.com/137017937. Our evaluation shows that ace extracts code with high accuracy, enabling deep indexing of video tutorials.
Abbreviations and Notations

**OCR** — Optical Character Recognition  
**LM** — Language Model  
**SLM** — Statistical Language Model  
**IDE** — Integrated development environment  
**ROI** — Region of interest  
**MSA** — Multiple Sequence Alignment
Chapter 1

Introduction

Programming tutorial videos teach programming by showing the computer screen as a programmer walks through pre-written code or writes code on-the-fly. Watching these is similar to training by pair programming, and provides an easy way to learn from the experience of others.

The number of programming tutorial videos on the web increases on a daily basis. Video hosting sites such as YouTube host millions of video lectures, other sites such as SlideShare and Coursera offer educational material on massive scale. Further, live streams of programming sessions are emerging as a valuable educational resource [1], producing even more programming video content.

**Goal** The goal of this work is to enable indexing of programming tutorial videos by accurately extracting the code that appears in the video. Accurate extraction enables both *searching for a video* by understanding its content as a whole, and also *searching inside a video*, allowing to find the specific part of a video that contains code of interest. The ability to search tutorial videos has the potential to revolutionize programming education, and allow programmers to find specific explanations during the development process. For example, imagine a world where the integrated development environment (IDE) includes an option to “find usages with tutorial walk-through”, where a programmer can find explanations for specific APIs and code pieces. A search can take the programmer directly to the part of the tutorial video that is of interest, thus effectively creating a “micro tutorial video” focused around a particular code or API. Indexing tutorial videos is also desirable...
Automatic understanding of general video content is a challenging problem [6]. In this paper, we target an easier special case of this problem — where video content is known to be textual, and is further known to be code in a given programming language.

**Challenges and Existing Approaches** Extraction of code artifacts from videos is a hard problem. There are two challenges that make extraction and indexing particularly difficult: (i) code in tutorials is typically written on-the-fly, with only parts of the code visible at each frame. To understand what constitutes a “code fragment” one needs to reason about the code across multiple frames of the video. (ii) optical character recognition (OCR) is not precise enough to produce quality results from videos [9], especially in the face of programming-specific settings such as varying font-sizes, colors, and annotations in an integrated development environment (IDE).

Although there has been a lot of work on automatic indexing of videos, and on extracting text from videos [8], most of the work has focused on detecting small amounts of text (e.g., captions, product labels, signs) in a video frame [9]. There has also been a lot of work on OCR postprocessing, where statistical language models [15] have been used to improve recognition in a document [18, 21], but these do not extend to videos nor to documents containing programs.

As a result, existing solutions for searching in programming tutorial videos mostly rely on user-provided meta-data.

**Our Approach** We present a framework for extracting code from programming tutorial videos. Our approach combines several ideas to achieve effective extraction, but the key ideas are: (i) Leverage cross-frame information to identify code fragments and improve accuracy of extraction, and (ii) Use statistical language models [15] at several levels to capture regularities found in code, and use them to improve the quality of code extracted from the video. Our approach combines base language models (LM) that capture regularities at the level of the programming language and are used across all videos, together with video-specific language models that capture regularities specific to each particular video. We use language models to enable correction at the token level, line level, and code fragment level. Our line-level model captures syntactic information about the structure of the line in addition to tracking token values. To train our base language
models, we use millions of code fragments obtained from GitHub and other repositories.

**Main Contributions** The contributions of this paper are:

- A novel approach for indexing programming tutorial videos by accurate extraction of code appearing in the video. Our approach uses statistical language models (LM) at several levels: tokens, lines, and fragments, to capture regularities in code that are later used to improve accuracy of extraction. A unique feature of our approach is the use of a language model over line-syntax to capture what constitutes a structurally valid line of code.

- Our approach leverages the common case of (at least partial) code repetition across multiple frames. By identifying and aggregating similar frames, we can handle the common setting in which code is written on the fly. By training a video-specific language model we handle cases where certain frames are particularly noisy, and cases where the text in a frame is partially obstructed (e.g., pop-ups).

- An implementation of our approach in a tool called **ACE** and an experimental evaluation on 40 real world programming tutorial videos. Our evaluation shows that **ACE** extracts code with high accuracy.
Chapter 2

Overview

In this chapter, we provide an informal overview of our approach using an example.

Consider a programming tutorial video such as “Android Tutorials 61: Broadcast Receivers” (https://www.youtube.com/watch?v=b7P6XisSoog) in HD 720p quality. Given this video, the goal of our approach is to extract and index the code that appears in the video, while ignoring parts of the video that do not contain code. In this simple example there is a single code fragment that can be fully observed in a single frame, but we would like to extract code even when different portions of the code appear in multiple and different frames across the video.

Further, we would like to be able to navigate the video to the point where the writing of a certain snippet has started, even in the common case where the code is written on the fly, and not all words of interest appear in the starting video frame. The ability to navigate to the point of the video where a certain explanation begins is important if we want to present a programmer with an explanation on how a certain functionality is used. The ability to navigate between different code snippet that may appear in the same video is important for presenting the programmer with a semantic table of contents for a video.

The ability to index and search a video relies on the ability to accurately identify the code in each frame, and also identify which frames contain related code. To address these challenges, ace first uses tailored video and image segmentation techniques followed by advanced OCR to extract text from video frames, and then uses our algorithms to process the raw frames and
extract code. Fig. 2.1 shows an overview of our approach. Our approach uses statistical language models to accurately extract code from the video. A base language model is first trained on over a million code snippets to capture regularities in the programming language. This base model is generic and is used as a baseline across all videos. To account for information that appears in the video itself, we train additional video-specific language models that help correct one frame based on the information in other (related) frames.

2.1 Extracting Raw Text

The first step in our approach is to extract raw text from the video. This is done using the following steps.

**Video Segmentation** To identify the frames of interest in the video out of tens of thousands frames in a typical video, we first preprocess each frame using image processing operations to prepare it for later stages (e.g., image resizing, quantization, and smoothing). We then choose frames based on sampling, and discard frames that do not contain code (based on frame segmentation).

The example video is 8:09 minutes long, at a frame rate of 30 frames per second, leading to a total of 14,670 frames. After sampling the video uniformly and filtering frames that do not contain potential code, we keep 50 frames.

**Frame Segmentation** To identify the region of interest (ROI) in a frame, we identify the main editing window of an IDE. Identification is based on seg-
Text Extraction: We use OCR techniques to extract the text from the ROI (main editing window identified in frame segmentation). This text is typically very noisy and the next phases of our approach are required to turn it into readable code. We refer to the text extracted from each frame as the raw text of the frame. Examples for snippets as extracted from different frames using OCR are shown in Fig. 2.3. Our goal is to extract “the best” code snippet that corresponds to the extracted text.

2.2 Extracting Code from Raw Text Frames

Intuitively, the noisy text extracted via OCR implicitly defines a space of code fragments: all the fragments that can be constructed by “cleaning” and combining the extracted text. Our goal is to find the most likely code
public class AirplaneReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Toast.makeText(context, "Airplane mode Changed", Toast.LENGTH_SHORT).show();
    }
}

Figure 2.3: Noisy raw text buffers as extracted from video frames by applying OCR.
fragments that can be constructed from the raw text of all frames. Towards that end, we leverage two high-level ideas.

**Cross-frame Information** We first find extracted text buffers that contain similar text. The idea is to find frames with some overlap, and use the aggregate information in the frames to improve the extracted text. We use these similar frames to train video-specific language models that is used to improve the extracted text in each frame based on the information in other (related) frames. As we show in the experimental evaluation, even when the video is sampled sparsely, there is still repetition in the text that appears across frames. We use this repetition to overcome noise and obstructions that vary between frames.

**Finding likely code using statistical language models** We then use three different statistical language models to find the most likely code fragment that can be extracted from the set of similar frames. These models are shown in Fig. 2.1 as *token model*, *line model*, and *fragment model*. As we explain later, while the *token model* captures token values (as commonly done in statistical language models), the *line model* captures the syntactic structures present in a line of code, and the *fragment model* captures relationships between lines. In other words, the line model and fragment model capture syntactic information and not lexical information. Further, the line model captures the common syntactic structure of a *line of code*. This is important because OCR works via line segmentation, and handles every line separately. This may lead to entire lines that are garbage due to noise (e.g., the first line in Fig. 2.3(b)) and should be discarded, or lines where errors can be corrected at the line level (e.g., the first line of Fig. 2.3(a)) which we explain next.
package com.te.broadcasttutorial;
import android.content.BroadcastReceiver;
public class AirplaneReceiver extends BroadcastReceiver{
    @Override
    public void onReceive(Context context, Intent intent){
        Toast.makeText(context, "Airplane mode Changed", Toast.LENGTH_SHORT).show();
    }
}
Chapter 3

Background

In this section, we provide basic background and definitions used throughout the paper.

3.1 Optical character recognition (OCR)

Optical character recognition is a method which converts images of text to a plain text stream. OCR is used to convert text written on checks, invoices, etc. In this paper we use OCR to extract text from video frames. There are a number of main essential stages which are implemented in most OCR programs. First, a de-skew is implemented to make sure the text is as perfectly horizontal as is could be. Second, some sort of filter is used to remove positive and negative spots, and furthermore to smooth edges. Then, the algorithms usually removes non-glyph lines and boxes. Afterwards, paragraphs, lines and words are detected. Finally, character recognition is implemented. There are two basic types of character recognition algorithms.

Pattern recognition: in this type of algorithm the computer learns several fonts and recognizes letters by matching them to the learnt options. This means that whenever we wish to recognize a new font, we first have to teach the OCR program to recognize letters written in the specific font. This is the type of OCR algorithm we use in this paper.

Feature detection: is also known as intelligent character recognition (ICR). Instead of recognizing the complete pattern of a character, detect the individual component features (angled lines, crossed lines, etc.) from which the
character is made. And thus, to estimate which letter we observe.

3.2 Statistical language models

Statistical language models have been used to model the regularities in natural languages and improve the performance of problems such as speech recognition, statistical machine translation, optical character recognition, and others [28]. In this paper, we use regularities found in code in several levels: token level, line level and fragment level. The token level and fragment level models are N-gram language models.

**N-gram language models** This language model assumes the probability to observe $x_i$ is based only on $x_{i-(n-1)},...,x_{i-1}$. This means that the token model assumes the probability to observe a token in position $i$ within the line is based on the tokens in positions $i-(n-1),...,i-1$ within the same line. And in the fragment model, the probability to observe a line in position $i$ within the line is based on the lines in positions $i-(n-1),...,i-1$ within the same fragment. The N-gram model will assign values in a way that maximizes Equation 3.2.

$$\prod_{i=1}^{m} Ngram(x_i|x_{i-(n-1)},...,x_{i-1}) \quad (3.1)$$

In this paper we use Bigram model, therefore the probability to observe a token is based only on the previous token. We also use the Unigram model which simply computes the partition of tokens within the language. We use the Bigram model in the fragment level, therefore the probability to observe a line is based only on the previous line. We assign values to tokens during the "Forcing Line Structure" step, based on the Bigram model over tokens. We assign values to tokens in a way that maximizes Equation 3.2. The equation maintains $V_x$ as the value assigned to token $x$

$$\prod_{i=1}^{n} Bigram(V_i|V_{i-1}) \quad (3.2)$$

13
3.3 Alignment

Sequence alignment is a method for matching sequences, mainly used in bioinformatics field of research to match DNA, RNA or protein. It is usually aimed at identifying regions of similarity that may be a cause of functional, structural or evolutionary relationships between the sequences. This method is implemented by putting aligned sequences into a matrix such that similar characters are put in successive columns, and gaps are inserted between the residues. In this paper, we use alignment to find corresponding lines and tokens (Section 5.1 and Section 5.1.1).
Chapter 4

Model

In this chapter, we provide a high-level formal model of the problem.

4.1 Likely Corrections of Programming Tutorial Videos

Given a programming tutorial video V, our goal is to extract all code fragments appearing in V, and record the frames in which they (and their parts) appear in.

**Frames and Raw Text Buffers** Formally, we define a video to be a sequence of frames $F_1, \ldots, F_N$. Using image processing and OCR to extract the text from each frame produces a sequence of raw texts $R_1, \ldots, R_N$ where each raw text is a buffer of characters.

Due to video compression, user interaction (pop-ups, completions, cursor movements, etc.), and OCR artifacts, the raw text is typically noisy, and is subject to various errors. Formally, we consider token errors, and line errors. In both cases, the error could be partial (e.g., part of the token is incorrect, or the entire token is redundant).

Intuitively, the raw text extracted from a frame implicitly defines a space of code fragments—all code fragments that can be extracted from the text by applying correction operations. The challenge is to find the most likely code that can be constructed from the raw texts of all frames.

**Token Variables** In a raw text buffer, we tokenize the text (split it into tokens) based on the target programming language lexical rules. We then
Figure 4.1: The symbolic version (where tokens are turned into variables) of the text buffer from Fig. 2.3(a).

treat tokens in the raw buffer as having an unknown value, and would like to determine the probability of each token having a certain value.

For example, Fig. 4.1 shows the variables in a single symbolic raw buffer with their values as seen in this specific text buffer. Variables are numbered in a positional manner by the line number and the token number in the line. When numbering variables across frames we add the frame number $k$ such that $v_{i,j}^k$ corresponds to the $j$-th token in line $i$ of the $k$-th frame. We denote by $r_{i,j}$ the raw value read at the position from the raw text buffer.

For example, the entry $\langle v_{i,9}^1, asttutorial \rangle$ means that the 9-th token on the 1-st line of the 1-st frame contained the value asttutorial.

We denote by $P(v_{i,j}^k = w)$ the probability that the $j$-th token in line $i$ of frame $k$ should have the value $w$. Our goal is to pick the most likely assignment of tokens for a given frame. In cases where we have an observed value, we can cast this question as $P(v_{i,j}^k = w \mid a)$ where $a$ is the actual observed value.

One way to compute the most likely token assignment is to pick token values using a base statistical model $B$ for common phrases in the programming language. For example, considering Java, and a trigram model, the sentence public static void is quite common. The base model $B$ repre-
sent common phrases in the programming language, and is not specific to the current video being analyzed. As expected, because the base model is generic, it cannot help predict values of tokens that are specific to the video itself (e.g., user class names, method names, variable names, etc.). However, even a unigram base model can be useful in simple cases where the observed value is close to a common value. For example, \( P_B(v_{1,1} = \text{package}|\text{package}) \) is very close to 1.

Another approach is to pick token values that are likely for the current frame. That is, to maximize \( P(v_{i,j}^k = w \mid F_k) \) for a given frame \( F_k \). However, a token value that is most likely in one frame may not be the most likely when considering all frames. In particular, some frames may contain little text (for example, when the first lines of the code are being written on-the-fly), and thus do not provide sufficient context for prediction. Considering a frame together with other corresponding frames may provide a significantly better prediction.

**Training a video-specific Language Model** Given a sequence of symbolic raw buffers, our goal is to construct a set of code snippets by assigning likely values to each token variable, possibly omitting certain tokens, and also potentially adding others. Towards this end, we construct a language model \( M \) that captures common sentences in the video, and use the model \( M \) (together with the base model \( B \)) to pick likely token assignments for the video. Given the limited structure of the problem, we proceed by computing the most likely assignments for each line, and use those to compute the most likely assignment for the entire code fragment. The first step in the process is to find correspondence between token variables of different frames, such that we can collect the possible values for each token variable across frames.
Chapter 5

Our Approach

In this chapter, we describe the steps taken by our approach to extract code from a given video. Algorithm 1 shows the algorithm used to extract code from a sequence of text buffers. The text buffers are the ones extracted from the video using a standard OCR tool and are order by time of appearance in the video.

The algorithm maintains:

- $C$ - a partition of text buffers into (disjoint) sets.
- $L$ - a mapping from a line in a text buffer to a set of corresponding pairs of line and text buffer.
- $T$ - a mapping from a token variable (position in a line in a text buffer) into corresponding token variables.
- $vtv$ - a distribution of possible values for a token variable.
- $P_v$ - mapping from a token variable to its possible values
- $\mathcal{V}$ - a mapping from a line number to a set of sequences of token variables forced that were forced into line structure

First, we describe the algorithm used to gather for each token variable all of its possible values (Section 5.1). Then, we describe the algorithm used to force structure on each line. Hence, to find the most likely assignment for each lines token variables based on possible values and on a statistical language model of line structureSection 5.2. Finally, we describe the algorithm
used to find the most likely fragment structure for each fragment based on
the line assignments and a Bigram model over line structures Section 5.3.

Algorithm 1: Extracting code from a sequence of text buffers.

**Input:** A sequence of text buffers $B_1, \ldots, B_n$

**Input:** Unigram Model UM, Bigram Model BM, Line Structure Model LSM, Bigram model over Line Structures BSM

**Output:** A set of code snippets $S_1, \ldots, S_m$

1. // Find possible values of token variables
2. $C = \text{compute corresponding frames}(B_1, \ldots, B_n)$
3. for $c \in C$ do
4.   $L = \text{compute corresponding lines in } c$
5.   for $l \in L$ do
6.     $T = \text{compute corresponding tokens in } l$
7.     for $t \in T$ do
8.       $\text{vtv} = \text{compute distribution of token values based on } T$
9.       $\mathcal{P}_V(t) = \text{close}(\text{vtv}, \text{UM})$
10. // Force line structure
11. for $c \in C$ do
12.   for $0 \leq i \leq \text{lines}(c)$ do
13.     $\mathcal{V}(i) = \text{compute assignment to token variables of line } i \text{ based on } \mathcal{P}_V,$
14.       BM and LSM
15.     // Find most likely fragment structure
16.     $s = \text{compute most likely fragment structure based on } \mathcal{V} \text{ and BSM}$
17.     $S.\text{add}(s)$
18. return $S$

5.1 Finding Possible Values of Token Variables

Finding Corresponding Frames

In this section we describe the algorithm used to group together corresponding text buffers. Since our goal is to extract code from a video and not from a single image, we can leverage repetitions of the same code in different frames across the video. We would like to leverage repetition of the code even if only parts of the fragment are repeated. To leverage repetition across frames, we first need to collect all text buffers that have been extracted from frames that contain similar code. In most cases, these are neighboring frames, but in some cases they may be frames that appear far apart. For example, when
the programmer jumps between several different code fragments, or when
the code in some frames is blocked by a popup window.

Because a text buffer corresponds to the single frame from which it was
extracted, we often use the terms “frame” and “text buffer” interchangeably.

Finding corresponding frames is similar to document clustering, but in
our setting, we can take advantage of several observations:

1. *Temporal Proximity* usually indicates correspondence

2. When a frame should be added to a set of corresponding frames it
   is because of its similarity to the last added frame due to Temporal
   Proximity

Aside from these observations, there are several challenges in finding cor-
responding frames that make it hard to use standard document clustering
algorithms. One of the main challenges is the fact that in the beginning,
most of the code fragments are very similar (as the programmer starts to
write them), and clustering without taking temporal proximity into account
is likely to put them in the same cluster.

*Code fragment beginnings are very similar:* In the beginning, most of the
code fragments look the same. But, whenever the author creates a new code
fragment she usually adds enough code lines to differentiate it from other
code fragments, before starting a new fragment. While standard cluster-
ing algorithms will group together all the beginnings of code fragments, by
adopting assumption 2 we can keep the text buffer disjoint.

The algorithm for partitioning the video to corresponding frames is de-
scribed below. After we partition the frames into sets, we sample additional
frames from the video around the beginning and ending of each set of cor-
responding frames, and apply the algorithm again. We sample additional
frames to handle the case where the first/last frame in a set do not have suf-
cient cross-frame information (they are the last/first frames of a different
fragment), and since we want to make sure sets were not separated due to
missing frames.

*Parameters for clustering algorithms:* Another, slightly less significant chal-
lenge is the choice of parameters in standard clustering algorithms. Some
clustering algorithms require the number of clusters in advance. Density-
based algorithms require some measure of expected density for a cluster, etc.
While we could have used some adaptation of these clustering algorithms, the importance of temporal proximity led us to use our own partitioning algorithm.

**Algorithm for partitioning a video to corresponding frames** This algorithm aims to collect corresponding frames. That is, to partition all frames into disjoint sets, where frames in each set are related to each other. We add a frame to a set based on a distance metric between frames, and based on its temporal proximity to other frames.

The algorithm iterates over the sequence of frames and tries to add the next frame to one of the existing sets. If no set was found as a match it creates a new set. Trying to match a frame to a given set is done by calculating the distance between the frame to the $N$ frames recently added to the set. If the distance to either of the last $N$ added frames is sufficiently close, we add the frame to the set. If the frame was found to be a match to more then one set the algorithm merges the sets and add the frame to the new unified set. We remove sets that do not contain a sufficient amount of frames since this sets are probably garbage. The distance metric used is Edit Distance divided by the length of the longer text. Edit Distance metric measures the amount of edits (e.g remove, add or replace a character) needed to be done in order to transform one text into the other.

The algorithm has two parameters: $N$ - the number of recently added frames that should be considered when comparing a frame to a set, and $\epsilon$ - a threshold on the distance metric to determine when two frames are considered as similar (normalized to the range $[0, 1]$). In our experiments, we simply use $N = 1$ and $\epsilon = 0.4$ meaning that we only compare to the most recently added frame in each set, and are quite liberal in our requirement on similarity between frames.

**Finding Corresponding Lines**

In this section we describe the algorithm used to group together corresponding lines in a specific set of corresponding frames. After we created sets of corresponding frames in Section 5.1 we wish to leverage the cross frame information. Therefore, we need to gather for each token variable $v_{i,j}^k$ all of its corresponding token variables $v_{i',j'}^{k'}$. As a reminder, $k$ refers to the frame number, $i$ refers to the line number within the text buffer as extracted from
frame \( k \), and \( j \) refers to the token number within line \( i \). The next step in the process of finding corresponding tokens is to gather corresponding lines. Corresponding lines are lines from different frames that seem to be occurrences of the same line of code. Hence, finding for each pair of \( k, i \) all of its corresponding pairs \( k', i' \). We use \( \sim \) to denote the correspondence relation between lines from different frames.

In addition to the observations regarding temporal proximity made in 5.1, we can rely on the following assumptions:

1. OCR process on an image does not duplicate lines.
2. OCR process on an image keeps the line order as it appears in the picture.

Using these assumptions, we conclude the following about the correspondence relation \( \sim \):

- Based on assumption 1, for each pair of \( k, i \) and \( k' \) there can be at most one \( i' \) for which \( k, i \sim k', i' \). That is, given a line in a frame, there is at most a single line corresponding to it in another frame.

- Based on assumption 2, for a specific frame number \( k \) and two different line numbers \( i_1, i_2 \) within \( k \), whereas \( i_1 < i_2 \), there can not exist a frame number \( k' \) and two line numbers \( i_1', i_2' \) within \( k' \), whereas \( i_1' < i_2' \) that supplies both (i) \( k, i_1 \sim k', i_2' \) and (ii) \( k, i_2 \sim k', i_1' \). That is, if two lines appear in some order in a given frame, their corresponding lines in a another frame appear in the same order.

- This relation is transitive, if \( k_1, i_1 \sim k_2, i_2 \) and \( k_2, i_2 \sim k_3, i_3 \) then \( k_1, i_1 \sim k_3, i_3 \). Hence, corresponding lines are disjoint sets.

**Algorithm for finding corresponding lines** The problem of finding corresponding lines across frames can be viewed as a problem of Multiple Sequence Alignment (MSA). Finding the optimal solution for MSA with \( n \) sequences is known to be NP-Complete. Therefore, we use a standard progressive alignment construction method. A progressive method first aligns the most similar sequences and then adds successively less related sequences. Our heuristics for which text buffer to align next is based on the temporal proximity assumption, the next text buffer to align is the next text buffer.
in the sequence. We remove sets that do not contain a sufficient amount of lines since this sets are probably garbage.

5.1.1 Finding Possible Values for a Token

In this section we describe the algorithms used to collect the possible values for each token variable. The values for a token variable are collected from: (i) all raw values as seen in the video and (ii) a statistical language model of the programming language. The raw values of token variable, mentioned in (i), are the raw values of the corresponding tokens.

The first step is to find these corresponding tokens. After we found corresponding lines in Section 5.1 we can finally find corresponding tokens. The algorithm is described below. After finding the corresponding tokens we can collect all raw values as the group of possible values “as seen in the video”.

But, there can be a case in which the real correct value is a token in the programming language that was never correctly extracted from the video. To address these cases we use a unigram statistical language model. The unigram model captures the distribution of tokens in the language. We add to the possible values group, all the tokens from the model that are similar to the values already in the group. That is, we check whether known common token values (from the language model) are close enough to current values in the group, and if so - add them as possible values.

**Algorithm for finding corresponding tokens** Given a set of corresponding lines we wish to find for each token variable all of its corresponding token variables. In this step we do not need to rely on Temporal Proximity since we have already gathered all similar lines. Some of the lines may be partial because of popups or because they have been captured while the author was writing them. But apart from partial lines, all corresponding lines in a set are similar to one another.

The assumptions made in 5.1 holds at the token level as well. That is, OCR does not duplicate tokens nor does it change the order of tokens within a line. Therefore, our goal is to split the corresponding lines into tokens and align them.

We split each line to tokens using a modified version of the programming language lexer, we modify the lexer so it won’t crash due to OCR mistakes.
For example, after the original lexer identifies the character ”, it enters comment lexing mode and expects to identify another ”, but as a result of the OCR process this character can be either missing or redundant.

After splitting lines to tokens, our goal is to align all corresponding lines. Normally we deal with tens of lines in each group of corresponding lines, therefore we use an adapted version of a progressive multi sequence alignment algorithm. First we compute the distances between all lines, and start by aligning the most similar two lines. Then we align the lines, line by line to the already aligned lines. The next line to be aligned is the line that is the most similar to any of the already aligned lines. Finally, we remove sets that do not contain a sufficient amount of tokens since this sets are probably garbage.

5.2 Forcing Line Structure

In this section we describe the algorithm used to find the most likely line structure for a sequence of token variables. Our goal is to find the most likely assignment for each token variable based on all the token variables in the same line, possibly omitting certain tokens, and also potentially adding others. Hence, we need to build the most likely line from the sequence of token variables. We use a statistical language model that captures the distribution of line structures in the programming language. We provide details on the training phase of the line structure model in Chapter 6. We choose the most likely assignment for each token by trying to force structure on the entire line. The result of this process is an assignment for each token variable in the line, while token variable can be removed and new token variables can be added to the line according to the chosen line structure. The algorithm used to force a line structure on a sequence of token variables is described in Section 5.2. We keep for each line, the group of line structures that require the least of changes.

**Forcing Structure on a Sequence of Token Variables** Given a line structure and a sequence of token variables with their possible values and a line structure our goal is to assign a possible value to the token variables, while possible adding and/or removing token variables. A line structure is a sequence of grammar symbols.

We need to assign a token variable to each symbol according to the
language constraints. For example, to the terminal symbol ‘(’ we can only assign a token variable with a possible value of ‘(’.

According to these constraints, we choose the assignment for each token variable. But a token variable can be matched to more than one symbol, therefore, we choose the assignment that maximizes the amount of matched token variables. Token variables that didn’t match any of the symbols are removed, while symbols that didn’t match any of the token variables are added.

For obvious reasons, if the added token variables are not assigned with any of the programming language operators or keywords, the line structure cannot be forced on this sequence of token variables. While certain token variables can be assigned based only on the constraints, certain variables, identifiers for example, can still be assigned with more than one option. We assign values to token variables based on a bigram model over tokens in the programming language, we provide details about the training phase in Chapter 6. We assign values to tokens in a way that maximizes Equation 5.1. The equation maintains \( V_x \) as the value assigned to token \( x \)

\[
\prod_{i=0}^{n-1} \text{Bigram}(V_i|V_{i+1})
\]  

(5.1)

5.3 Fragment Level Correction

In this section we describe the algorithm used to create the most likely code snippet from the lines constructed earlier. As described in Section 5.2, the result of forcing line structure on a sequence of token variables, is a set of possible lines. Our goal is to choose for each line, its most likely line structure in context. For example, from the sequence of tokens + @ Override we can create two line structures:

- + Override
- @ Override

both of which require the same amount of modifications.

Since the amount of changes is the same, the naive way would be to choose the assignment based its probability in the line structure model. This naive way does not consider the context. We would like to choose the
most likely assignment for each line based not only on the line structure but on other lines in the same fragment. In order to assign for each line it’s most likely structure in context we use a bigram model over line structure, we provide details about the training phase in Chapter 6. We assign structures to lines in a way that maximizes Equation 5.1. The equation maintains $V_x$ as the structure assigned to line $x$.  


Chapter 6

Implementation

We implemented our approach in a tool called ACE. ACE is implemented as a series of utilities that train statistical language models on a large number of code samples and extract code written in Java, from programming tutorial videos which teach how to write code for the Android platform. ACE is implemented both in Java and python and depends on OpenCV [4] for frame extraction and image processing, on Tesseract OCR library [2] for text extraction, on scikit-learn [13] for clustering, and on ANTLR [12] for parsing and lexing.

6.1 Android Framework

Programming tutorial videos explaining how to write code for the Android platform are known for their popularity on tutorial video hosting sites. In android tutorial videos one can observe more then a single programming language, for example, files written in xml are often used by the programmer to represent the application’s layout. This fact obligates ACE to differ code written in Java, the target language, from other programming languages.

6.2 Extracting Text Buffers from a Video

Extraction of text buffers requires several steps:

**Video segmentation** We extract frames from the video by uniformly sampling the video. In the future, we plan to employ more advanced segmenta-
tion techniques, but these are orthogonal to our approach.

**Frame segmentation** We identify the region of interest (ROI) by first finding all the contours within the image. We use Canny edge detection [7] and we apply a standard tool for finding contours. Then, we choose the smallest counter that seems to cover most of the code in the image as the ROI. Finding the main editing window for every frame is computationally prohibitive and in times impossible due to limitations of image processing techniques. Therefore we often find the main editing window in one of the frames we sampled and use its position to extract code from all frames.

For each frame we identify pop-ups as contours that contain non-code text (using a simple classifier), and we mask their content in the image to diminish their influence on the OCR result.

**Text extraction** We extract a text buffer from the main editing window using the Tesseract OCR library. We use OCR with line-level segmentation, such that the extraction returns a sequence of lines.

### 6.3 Differing the Target Language

After extracting text buffers from a video, ace must identify text buffers that contain potential code written in the target language. We chose to extract code from Android programming tutorial videos, therefore we wish to identify text buffers that contain potential code written in Java. A line of code in Java mostly ends with one of these symbols: ; { }. ace uses a simple custom made filter that checks if the majority of lines in the text buffer contain at least one of the Java symbols ; { }.

### 6.4 Language Models: Training Phase

Millions of code samples that using Android libraries were obtained from GitHub and other repositories, and were processed into three statistical language models.

**Token model** we transform every line of the snippet to a sequence of token using the programming language lexer. We process the sequences of tokens into Unigram and Bigram models over tokens.

**Line structure model** we transform each code snippet into its Parse Tree.
We use the leaves of the parse tree and split them according to the lines in the code snippet. Therefore we obtain a set of sequence of leaves per each code snippet. We manipulate the leaves that represent identifiers according to the programming language conventions. For example, in the Java programming language, identifiers that start with an uppercase character are classes, interfaces and constants names while variables and methods names start with a lowercase character. A sequence of leaves is our line structure. The model computes the distribution of line structures in the programming language.

**Bigram over line structure model** we transform each code snippet into sequence of line structures as we did for the line structure model. The model computes the probability to observe the next line structure given the current line structure.
Chapter 7

Evaluation

In this chapter, we report the results of our experimental evaluation.

7.1 Methodology

To evaluate our approach, we consider 40 Android programming tutorial videos in HD 720p quality taken from YouTube. We focus on HD quality videos because it is very popular among video hosting sites, and there is a huge number of tutorials available in this quality. Even with HD quality, direct text extraction is insufficient for code indexing (as shown in Fig. 2.3). It is important to note that our techniques apply to video of lower (and higher) quality. For SD videos, the approach is typically precise enough to pick up keywords, but not enough for extracting complete code fragments with high accuracy. Our techniques for consolidating code across frames, handling code that is written on the fly, obstructions due to pop-ups, etc. are useful even with perfect extraction from a single frame.

We watched each video and manually extracted the code that appears in it. Transcribing the code from all videos took several days of work.

*Code in a Video* A programming tutorial video may contain multiple code snippets. Further, in many videos, even a single snippet does not appear as a whole in a single frame. When extracting the code from a video, we *manually* merge all code that belongs to the same file into a single snippet. Of course, when there are multiple *different* snippets in a video, we do not merge them. This manual extraction of code from the video gives us a
“ground truth” to compare to.

Comparing Code vs. Comparing Text Given the manually extracted “ground truth”, it is not clear how to compare to it. Should the automatically extracted code be compared textually, for example, using character-level diff? Such measurement misses the fact that we are dealing with code.

We are interested in comparing the code extracted from the video and not just text. Therefore, we evaluate the quality of our results by comparing partial parse trees from the extracted code to the parse tree of the manually extracted code. The parse tree of the manually extracted code may be partial as well, since the code snippets in the video may be partial.

Note that we cannot assume that the extracted code can be parsed as a whole. Further, even a single line of the extracted code might not be parsable. However, since we enforce a valid line structure on each line, we already have a partial parse tree for each line.

Our comparison is therefore based on comparing the partial parse trees obtained from the extracted code, with the parse tree obtained from the manually extracted code. We measure both coverage and precision of our extraction by comparing tokens at the leaf-level.

Video Sampling Method We use uniform sampling of the video and set a minimum number of samples. The fixed sample rate is 1 frame per 5 seconds, and the minimum number of samples is 200. Therefore we chose the minimum between the fixed sample rate and a calculated rate that will lead to the minimal number of samples, thus guarantying the minimal number of samples.

7.2 Results

Table 7.1 shows the programming tutorial videos we have used as benchmarks, the number of code snippets extracted manually from the video, and the number of snippets extracted by ace. The table summarizes the results of two sampling methods: fixed sampling, and relative sampling, which we explain below. For each technique, the column #Fr shows the number of extracted frames, the column D the number of automatically detected snippets, and the column E the number of additional detected snippets due to over-splitting of snippets (where we failed to merge two snippets despite
Table 7.1: Programming tutorial videos and number of extracted snippets using fixed sampling and relative sampling methods. \( \#\text{Fr} \) is the number of extracted frames that contain potential code, \( S \) is the number of actual snippets, \( D \) is the number of detected snippets, and \( E \) is the number of extra snippets due to over-splitting.

<table>
<thead>
<tr>
<th>Name</th>
<th>Duration (M:S)</th>
<th>( S )</th>
<th>( #\text{Fr} )</th>
<th>( D )</th>
<th>( E )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Android Eclipse.learn</td>
<td>1:18</td>
<td>1</td>
<td>170</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>30 - Color of TextView</td>
<td>3:56</td>
<td>1</td>
<td>201</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 23</td>
<td>4:06</td>
<td>1</td>
<td>173</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Button</td>
<td>4:46</td>
<td>1</td>
<td>118</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>How To: Android</td>
<td>4:55</td>
<td>1</td>
<td>109</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>How to stream video</td>
<td>5:22</td>
<td>2</td>
<td>133</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorials 62</td>
<td>6:15</td>
<td>3</td>
<td>68</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android App. Dev.</td>
<td>6:59</td>
<td>1</td>
<td>168</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Android Tutorial 64</td>
<td>7:00</td>
<td>1</td>
<td>54</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 19</td>
<td>7:28</td>
<td>3</td>
<td>143</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 29</td>
<td>8:03</td>
<td>1</td>
<td>104</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 61</td>
<td>8:09</td>
<td>1</td>
<td>50</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 63</td>
<td>8:45</td>
<td>2</td>
<td>84</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Chronometer</td>
<td>9:05</td>
<td>1</td>
<td>164</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>How to read file</td>
<td>9:16</td>
<td>1</td>
<td>143</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Android Create Menu</td>
<td>9:28</td>
<td>5</td>
<td>105</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 7</td>
<td>9:47</td>
<td>2</td>
<td>137</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 18</td>
<td>9:53</td>
<td>3</td>
<td>92</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Fragment Tutorial</td>
<td>9:54</td>
<td>4</td>
<td>109</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorials</td>
<td>10:30</td>
<td>1</td>
<td>130</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 43</td>
<td>11:19</td>
<td>1</td>
<td>128</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 38</td>
<td>11:19</td>
<td>2</td>
<td>141</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 55</td>
<td>11:46</td>
<td>1</td>
<td>120</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 17</td>
<td>11:55</td>
<td>5</td>
<td>129</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Delete Selected</td>
<td>11:56</td>
<td>1</td>
<td>163</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Android Tutorial 13</td>
<td>12:02</td>
<td>2</td>
<td>154</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Web View</td>
<td>12:03</td>
<td>1</td>
<td>94</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Basic ListView Demo</td>
<td>12:54</td>
<td>1</td>
<td>135</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 48</td>
<td>13:16</td>
<td>1</td>
<td>99</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Alert Dialogs</td>
<td>14:07</td>
<td>1</td>
<td>109</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Android Tutorial 22</td>
<td>14:20</td>
<td>2</td>
<td>181</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>App in 15 minutes</td>
<td>14:39</td>
<td>1</td>
<td>80</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 5</td>
<td>15:06</td>
<td>1</td>
<td>124</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>File Read</td>
<td>15:19</td>
<td>1</td>
<td>161</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>41. Drag and Drop</td>
<td>15:28</td>
<td>1</td>
<td>156</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Android Tutorial 36</td>
<td>17:03</td>
<td>3</td>
<td>134</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 21</td>
<td>18:00</td>
<td>3</td>
<td>151</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 68</td>
<td>21:35</td>
<td>2</td>
<td>174</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Android Tutorial 72</td>
<td>25:14</td>
<td>2</td>
<td>260</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Android Pure Java</td>
<td>30:44</td>
<td>1</td>
<td>107</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
them actually being parts of the same program).

The duration of the videos ranges between 1 minute and 30 minutes. The number of snippets in each video is rather low (mostly 1) and up to 5 snippets. Note that it is often the case that we count a single snippet where different frames show different parts of the same overall code. This shows the strength of our approach in the ability to stitch together different parts of the program that appear throughout different frames of the video.

**Snippets in Each Video** Most of the videos we consider, contain a small number of code snippets around which the tutorial revolves. In most videos, the code is being written on the fly. This means that different frames almost always show different parts of the code (with some overlaps).

In some videos, such as MenuList, the author created 5 snippets, but 4 of them contain the exact same code, with the only difference being the class name of the containing class. These files are only 3 lines long and therefore were merged into the main snippet.

In Tutorial 22 (in short #22), there are two snippets, but one of them is only used as reference, where the authors jumps between the main code being written, and the reference snippet which is only shown for short periods. The extraction with the current parameters only extracts the main snippet.

In some videos, certain code fragments are shown for a very short duration (less than 30 seconds) where the code is being scrolled through and/or copied. In such cases, we may not be able to extract the code that is quickly glanced over (with the current parameters of the extraction).

**Split snippets:** The code snippet in 41. Drag and Drop, Android Tutorial 5, Android Chronometer, Delete Selected, , File Read and Android Alert Dialogs was split into two separate code snippets, creating one extra snippet.

**Garbage snippets:** Android App. Dev. had an extra garbage code snippet. Android Chronometer had 2 extra garbage code snippet causing the total of 3 extra snippets.

**Merged snippets:** In some videos two code fragments were merged into one code fragment, this happens in How to stream video and Android Create Menu.

**Coverage and Precision** We measure the coverage and precision of our extraction procedure by comparing the parse tree leaves of the manually extracted code to the extracted tokens and their grammar symbols. Coverage stands for precision of correct extracted tokens from the ground truth.
Figure 7.1: Coverage and precision for varying threshold.

tokens. Precision stands for precision of correct extracted tokens from the extracted tokens. For example, if the manually extracted code is:

```
public void onReceive(Context context, Intent intent){
    with parse tree leaves (ID_l, ID_u representing identifiers starting with a lowercase/uppercase letter respectively):
```

and the code as automatically extracted by ACE is

```
public void onReceive(Context context){
with parse tree leaves:
PUBLIC VOID <ID_l,"onReceive"> LPAREN <ID_u,"Context"> <ID_l,"text"> RPAREN LCBR RCBR
```

Tokens are identical if both the grammar symbol and value are equal. Therefore, we will report that there are 4 missing tokens, and 2 redundant tokens. Hence, the coverage for this line is 7/11, and the precision is 7/9.

Fig. 7.1 show the coverage and precision with different threshold values for what is the required number of occurrences for a line structure to be considered as valid in the model.

The threshold tries to capture what should be considered as a valid line structure in the language. If we only consider a line structure as valid if it occurs more than 10,000 times (the extreme points on the right), then we observe a coverage of 66.025 and precision of 80.15. The coverage is low since we try to force wrong line structures on valid lines causing tokens to receive the wrong grammar symbol.

On the other hand, if we consider any line structure that appeared more
than once as valid, we observe coverage of 81.125 and precision of 80.2. The coverage is high because we consider more line structures as being valid.

The precision remains the same since as the threshold increases the number of lines that don’t match any line structure increases and that causes both valid lines and garbage lines to be considered invalid.

Fig. 7.2 shows the percentage of different line structures which are considered valid as a function of the required number of occurrences in the model. As can be seen from the figure, the percentage of lines that appear more than once is 47%. Some line structures are very common and occur tens of thousands of times. Fig. 7.3 shows the ten most popular line structures in the model. Since programming for the android platform involves the use of plentiful libraries, one can observe that import statements are placed both in the second and fourth places. Further more, the massive use of inheritance causes the grate use of the statement appearing in the third, sixth and tenth places.

**Extraction Times** We measure the time required for processing each video. The time consists of several different components:

- extracting raw text from the video - this includes video processing, image processing, and OCR.
- finding corresponding frames
- finding corresponding lines
- finding possible values for tokens
- forcing line structure
- fragment level correction

The first phase requires passing over all frames of the video, sampling
them, performing image processing, finding the main editing window, handling pop-ups, and performing OCR. On a typical video, the first step takes between 5 to 10 minutes. Our implementation is not optimized, and there are many opportunities for parallel processing of frames that could make this process significantly faster.

All other stages of processing take between 10 to 30 minutes, on rare cases of extremely noise text it can take up to a couple of hours to find corresponding tokens. This is due to the naive alignment algorithm used and its non-parallel execution.

**Importance of Repetition** We observe that correction based on repetition of text between frames is effective. A typical line that appears in one frame repeats in between 3 to 70 other frames. Since videos typically show the code as it is being written, there are some lines that only repeat partially between frames (as the line is being typed). Note that when a line repeats between frames, it often appears in a completely different location on the screen, due to scrolling between frames, additional code being added above it, code being removed above the line, etc. Our approach is designed to find corresponding lines between frames to deal with such cases.

Figure 7.3: Top ten line structures.
Chapter 8

Related Work

**OCR Post-Processing** There has been a lot of work on OCR post-processing. OCR technology itself not precise enough to produce quality results when extracting code from videos [9], especially in the face of programming-specific settings such as code that is being written on-the-fly, varying font-sizes, colors, pop-ups, and annotations in an integrated development environment (IDE).

Tong and Evans [18] post-process OCR results using correction system based on statistical language modeling. They use letter n-grams to correct a given word and word-bigrams to correct a given sentence. Zhuang et al. [21] post-process OCR results using n-grams and Latent Semantic Analysis language models to achieve both syntactic and semantic information. Taghva et al. [16] remove “garbage strings” from the OCR text using generalized rules that can identify those strings.

Our approach can be viewed as an adaptation of OCR postprocessing that can: (i) leverage cross-frame information instead of working on a single document, and (ii) use statistical models of a programming language, instead of a natural language, and in particular, the common grammatical structure for a line of code.

**Extracting Textual Content from Videos** Merler and Kender [10] index presentation videos based on the text displayed in the displayed slides. They extract the text directly from the video and used changes in text to segment the video into semantic shots. They apply image processing techniques to deal with low image quality and did not use OCR post processing.
Yang et al. [20] index lecture videos by first segmenting the video using slide structure analysis. They use OCR to extract text from each frame, use spell checker to find correct extracted words, and then choose the extracted text containing the major amount of correct words. If two or more extracted texts have the same amount of correct words they choose the one with lower word count, and if they have the same number of words they combine all correct words as the index.

These approaches target the setting of a slide-based lecture, in which slide transitions typically lead to completely different text being displayed. Further, they rely on properties of natural language (e.g., a spell checker in [20]) to perform error correction.

**Error Correction in Parsers** Our approach is also related to automatic error correction in parsing [17, 3]. However, due to garbage lines, the direct application of such techniques seems to be rather challenging. In contrast to these techniques, our approach relies on statistical models that perform correction at the token level, line level, and fragment level.

**Statistical Language Model for Programming Language** Hindle et al. [5] suggest that code can be usefully modeled by statistical language models, and developed code completion engine for Java using trigram model that suggests a next token.

Raychev et al. [14] build a statistical language model over sequences of method calls created by applying static analysis on a large codebase. This enables effective code completion in the form of call sequences across multiple objects. In contrast to statistical models that capture call sequences, we use three different levels of models: one for tokens, one over syntax elements, capturing line structure, and one model that captures regularities between line structures to represent common structures of code fragments.

Tu et al. [19] shows that a lot of the regularities in human-written software are local. This observation is in line with our experience and is one of the reasons that our simple statistical models are sufficient in correcting code from programming tutorial videos.

Nguyen et al. [11] use a statistical language model enriched with semantic information. This allows them to capture regularities beyond what can be captured via pure lexical tokens. Our line-structure model is similar in nature to their enriched statistical model, but also captures what constitutes a valid line in the program. The focus on how code break into lines is critical
in the OCR setting, where segmentation is line-based and entire lines may end up being garbage lines due to noise.
Chapter 9

Limitations

During our experiments, we observed some limitations of our method, which we now describe.

Finding the main window: We rely on standard image processing techniques to discover the main editing window within an image. While these techniques are sufficient for most videos, there are videos, especially low quality videos, on which these techniques struggle to find contours. Our method relies on the fact that the amount of potential code in a single line is significantly larger than the amount of redundant tokens added due to noise. In an IDE, there are several areas with noticeable amount of text, for example the Package Explorer View that contains names of files used in the project. Text in these areas is usually small and multi-colored therefore the result of applying OCR on these areas is disproportionately noisy compared to the result of applying OCR only on the main editing window.

Unique fonts: While our method can handle splitting token errors and character modifications within a token, our method cannot handle cases in which two or more identifier tokens are consistently merging into one. Since our methods forbids adding an identifier token to a sequence of tokens in order to force a certain structure, we cannot construct the appropriate structure when tokens are merging. For example, in some fonts the character '(', is transformed to the character 'C' when applying OCR. A modified version of the programming language lexer, which we use to split lines to tokens, applied on an OCR result from video using this font will merge three tokens to one (e.g A(B→ACB). In order to extract code from these videos
using ace one should train the OCR tool on the particular font.

*Video quality:* When extracting code from low quality videos, standard image processing techniques struggle to find the main editing window and the standard OCR tool merges tokens and produce more noise than our method can handle.

*Glances at code:* When the author jumps from one code fragment to another without significant pauses we can not differentiate frames that contain code from fully noisy frames. Our method relies on the repetitiveness of code in a tutorial video and therefore treats singular code frames as noise.
Chapter 10

Conclusion

We presented a novel approach for extracting code from videos and indexing programming tutorial videos. Our technique extracts code directly from the video, and is based on the following ideas: (i) consolidating code across frames to improve accuracy of extraction, (ii) a combination of statistical language models for applying corrections at different levels, allowing us to perform corrections by choosing the most likely token, combination of tokens that forms a likely line structure, and combination of lines that lead to a likely code fragment in the language.

We have implemented our approach in a tool called ace, and used it to extract code from 40 Android video tutorials on YouTube. Our experimental evaluation shows that ace extracts code with high precision, enabling deep indexing of video tutorials.
Bibliography


בשלבה ראשונה לתחילה: בשלב ראשון, נָחֲמוּ אֶפְרִימִים: בָּשֶׁלֶב הַרְאָשָׁנוּ שָלָה התַּחֲלִיתָןָּו מַטְחָמִים מְרִיָּם
שָׁנְדַּמְּה הָכִּים כֵּלֵי הַקֵוֶסֶד הֵזוּ מְרִיָּם. כַּנֶּכֶל לַעֲלֹת אֶת הָכִּסֶּד הַחֶוְּרָה מַסְפְּרָה מְצַפְּרָה
בְּשֶׁלֶבֶס הַבָּאָם.

בשלב ב, פִּיעַת שָׁוָה הַחֲוֹמָתוֹ: גַּאָרִי שָׁחְלֵקַנְּוּ אֶת הָפִּירְמִים לַמְכָּבָּרִים מְרִיָּם הָחֲוֹמָתוֹ,
עֲלֵיֵל לַעֲלֹת שָׁוָה הַחֲוֹמָתוֹ שָׁוָה הַחֲוֹמָתוֹ בְּנֵי הָפִּירְמִים הַשָּׁוָה. שָׁוָה הָחֲוֹמָתוֹ הָנָּג הָשָׁוָה.
שָׁנְדַּמְּה כָּנָה מְפְעִלָּה שָלָה שׁוֹרָה.

בשלב ג, פִּיעַת עֲרֵכִים אֵפְרִיסִים לַחַלָּה: מָחַק הָשָׁוָה הָחֲוֹמָתוֹ אֶנֶּרִי מְרִיָּם
הָחֲוֹמָתוֹ שָׁוָה הָחֲוֹמָתוֹ בְּנֵי הָשָׁוָה. כֵּל סֵלֶכֶס הָחֲוֹמָתוֹ הָנָּג סֵלֶכֶס אֵפְרִיסִים
עֲבָר מְלַכֶּס, לָטַּה הָנֶנֶנֶס מִסֵלֶכֶס מְסַפְּרָה רָמֶכֶס הָמְלַכֶּס עֲשָר דָמִים מַבָּחָנָה
tכָּסֶלֶכֶס לְעָרֵכִים הָכִּים בֵּסֶס.

בשלב ד, פִּיעַת עֲרֵכִים אֵפְרִיסִים לַחַלָּה: עֲבָר כַל שׁוּרָה, מְצַנַּא כֵל מְלַכֶּס, אֶנֶּרִי סֵלֶכֶס אֵפְרִיסִים
לָמַלְחָה כֶּנֶנֶנֶס עֲבָר כַל מְלַכֶּס בָּשֶׁרֶת עֲרֵכִים, אוֹמֲסֵה יָבֵנֶנֶס כֵּל לָמַלְחָה
eחֲוָה אֶנֶּרִי מְסַפְּרָה לָשׁוּרֶה. אוֹנֶנֶס יָבֵנֶנֶס אֶנֶּרִי סֵלֶכֶס בָּשֶׁרֶת שׁוּרֶה אֶנֶּרִי מְסַפְּרָה
מלַכֶּס הָכִּים בֵּסֶס. אוֹנֶנֶס הָשׁוּרֶה כַל שׁוּרֶה אֶנֶּרִי הָמְלַכֶּס הָמְלַכֶּס לְעָרֵכִים לְעָרֵכִים
לָמַלְחָה בֵּסֶס כַל מְלַכֶּס.

בשלב ה, פִּיעַת עֲרֵכִים אֵפְרִיסִים לַחַלָּה: יַבֵּנֶנֶס מְסַפְּרָה מְבָנָנָה אֵפְרִיסִים עֲבָר שׁוּרֶה, אוֹנֶנֶס הָמְלַכֶּס
הָמְלַכֶּס הָמְלַכֶּס בֵּסֶס בֵּסֶס שׁוּרֶה. בָּשֶׁרֶת נַסְמַת בֵּסֶס הָמְלַכֶּס כֵּל בָּשֶׁרֶת
אֶנֶּרִי הָשׁוּרֶה הָשׁוּרֶה בֵּסֶס שׁוּרֶה בֵּסֶס כֵּל.
שלב האימון
בשלב האימון עלים מיצרים שלושה מודלים סטטיסטיים במחזור שוניות בכדי שיפתם את תוצאות ה钦ון ומודל ברמת השורות ומודל ברמת המסגרה. מודל ברמת השורות שובב יעיל משטח סטטיסטי עם פתרון מילוי שפת ה钦ון ומודל ברמת המסגרה. מודל ברמת המסגרה שובב יעיל משטח סטטיסטי עם פתרון מילוי שפת ה钦ון ומודל ברמת המסגרה. מודל ברמת המסגרה שובב יעיל משטח סטטיסטי עם פתרון מילוי שפת ה钦ון ובמחזור. מודל ברמת השורות ומודל ברמת המסגרה. מודל ברמת השורות ומודל ברמת המסגרה. מודל ברמת השורות ומודל ברמת המסגרה. מודל ברמת шורות ומודל ברמת המסגרה. מודל ברמת шורות ומודל ברמת המסגרה. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודלברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודלברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברامت שories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ברמת шories. מודל ברמת шories ומודל ב...
מטרות

מטרתה על מנוין עבודה ושל נובע עבודה זו היא לפתח מערכות עבורחולקי אוטיסטים של קוד מסרטונים עלילים לחנות. בהנהלת סרטיולים עלילים לחנות על המערכות של חולקי סרטיולים מחברת אוניברסיטתOCR על יעדים אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטתמחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטתמחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטתמחברת אוניברסיטת מחברת אוניברסיטת מחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניבرسיט

התקל

בעבודה זו היא מגוון מתוכן עיבוד לנובע עבודה זו היא לפתח מערכות עבור חולים אוטיסטים של קוד מסרטונים עלילים לחנות. בהנהלת סרטיולים עלילים לחנות על המערכות של חולקי סרטיולים מחברת אוניברסיטתOCR על יעדים אוניברסיטת מחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברתאוניב

הניתוח

בכדיتلדך קוד מתוכן סרטונים עלילולוחות להבנתו על המגדירות הבאות:

 sprzון פה סינון של פורים: ימי ה伊拉וי המועטים בשכם פורים. בעדר כל מדובר בחולק העריכה (IDE) המרכז בשכבות הפיתוח המשולב

חילון סתקס: חילון סתקס מתוכן פורים עליך שימשו בשכבות פיתוח של עיבוד תומת ויחוי

חילון קוד: תוחס חילוןמקל פורים עליך ימי חילון أفוטי כולם שיגור של בז וחבר

הפתרון מעבר

בעבודה זו היא מגוון מתוכן עיבוד לנובע עבודה זו היא לפתח מערכות עבור חולים אוטיסטים של קוד מסרטונים עלילים לחנות. בהנהלת סרטיולים עלילים לחנות על המערכות של חולקי סרטיולים מחברת אוניברסיטתOCR על יעדים אוניברסיטת מחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברת אוניברסיטתמחברתאוניב

ב
מטרות

 米טרית עלכל עבורה היא מחשת מערכות עבורה שקהל באטומטי של קוד מסרטונים לולימוד תחנת. בהנחת סוף קול לולימוד נצפת על המרכיבים לולימוד תקן קMounted הסר_io תקן העדifice על הקורא OCR בלעון עיבוד אוביישן של текסט שוחלך בדיל חליפה את הקורא

ведוכה מתכון הקקסט. על המ.protobuf להזון כדי בנגד מתכון הקקסט אוביישן י듭ע כדי שפפה

ולשנה מתכון הקקסט. בורב המתכון הקקסט כדי הופים מופייר פעים ביפים שיו

על המ避けתח לותר הקקסט אלא למכונת לולים קעד קעד יודי.

הליך

בנשבודו והאנ פיצים מערכות עבורה שקהל קוד מתכון מסרטונים לולימוד תחנת. המתרחק

שלכל מסך ערוצים בודי חלינה שקהל אפידמי, אשר הורו המרכז הזה שופמי מודלי פיסטייסים של שיפ במכונת הפסטים והמות

שופמי מודלי פיסטייסים של שיפ במכונת הפסטים והמות-

victim המודלי פיסטייסים של שיפ במכונת הפסטים והמות-

משתמש ביבב המוקים הקקסט כדי שושאר משגב קעד יודי.

התקני

בכדיל הלוח קד מתכון מסרטונים עלע לתחמוד על התראים הביאו:

глядית על ירוד: גיור הפרימיום המ ngồiים מתכון ערוצים אל피 פירימית.

סטטיסטיקה של פרימיר: גיור האור העונני מתכון פרימיר. בדרכ לכל מדובר בלהטריעה

(IDE) המרכז בטיבת הפשיטה המושבלת

תיקון מתכון פיסטייס של שיפ במכונת הפסטים של שיפ במכונת

תיקון开放式

תיקון מתכון פיסטייס של שיפ במכונת הפסטים של שיפ במכונת

גכן לתחפה קעד תקן.

פתורון

בעבורה והאנ פיצים מערכות עבורה שקהל קוד מסרטונים לולימוד תחנת. המתרחק

שלכל מערכה משלב מזון שלב שקהל מבוסס על הימולידים שיאומן.
תמצית

موتיצניה

כמות הסרטונים באינטרנט שמטרתם ללמדת програмирование עולה בכליום. אתר וישיתו לשתף סרטונים

Coursera מאחסנים מיליוני סרטונים ל)((((למדון frontal sistemas של תכנות עולמית מוסר, סרטונים ל }}}

למדון סיפות וכתיבת חידות ולהבנה של קוד שנכתב מראש.APPING תצפית בסרטונים אלה מתפתחת לעוף שליל الأمم של עיזי התוכן. היחלץ התוכן מתוך סרטונים אלה באפור אוטומטי דריץ עדבר מספר מוסר חן חוסר

לכלכישים דמויים.

יחלץ התוכן מתוך סרטון הוא走近ה מספריות. עבורה זו היא מתמדים במקודה ממקוד

שהסרטיון, סרטונים בהם יודיעי כי התוכן חוץسوقיאל משוער כלכ, יודיע שחרתקסט

והיון קוד המשמש תוכנות ידיו מראות.

גישות קיימות

יחלץ קוד סרטוניים היא走近ה מספריות. הדרד הנאות לתוכנות בועה זו היא העומת

OCR 작품י יпись לע פורימי בתוכנות סרטוניים.أس עמק, היא תכנית התרשים

שאני מודיעת יד בבדי המחלה קוד סרטוניים באופי יוקי. הבחין מוחרי הכשר

ממודיעים על בועה כונה רביי דלי נום, צבעים של((&(Popups) ושר מופעים

ב_preferences פיתוח של IDE. פורנוגרת קיימים חלפים סרטוניים אלה נסכמה עייק

על מטלא-โดยเฉพาะ שמספק עי מפסיק סרטון.

 Afro על פי שמצא את עבורה הר בתוכנ dépis או תכנית להודיק בתוכנות מוסר, רוב הובד

ההמדדה בחולץ כמות עקיס קשון חוץ תכנית וימי. ערוץ נסף שילג ונשון מהからない

בר היה עבורי erhalten של SignUp שין שים בודד סרטוניים לשפת שבדי

לשפר את אינט עשוי בקביעות, את מהכר זה לא נשיאת懷ה שסרטוניים ולא קביע

המכילים קוד.
המחקקר עשה בהנחיית פרופ' ערן יחב בפקולטה למדעי המחשב.

אנא מודע לסקונן על ההмиית הכספית והדיתה בהשכלתו. המחבר שוהיב לתחזוק.

אני קובל מילים מעכネット המחבר העבירתי של האותيج הארבאיה.
خيلז קוד מתוכן סרטיונים לישראל
tכנית

חיבור על מחקר

לשם مليיה חקקי של ה드리ות לקבלי התحواר
מגידים למיתרים יוזמיים המוחשב

 Shir Yeid}

הוגש לסנט הטכניון – מכון טכנולוגי לישראל
ницыית ההשלמה ± הייריל 2015

Technion - Computer Science Department - M.Sc. Thesis  MSC-2016-03 - 2016
הילוך קוד מבית סרטונים ללימודי תכנון

שינר יידר