Extracting Code from Programming Tutorial Videos

Shir Yadid
Technion, Israel
shiry@cs.technion.ac.il

Eran Yahav
Technion, Israel
yahave@cs.technion.ac.il

Abstract

The number of programming tutorial videos on the web increases daily. Video hosting sites such as YouTube host millions of video lectures, with many programming tutorials for various languages and platforms. These videos contain a wealth of valuable information, including code that may be of interest. However, two main challenges have so far prevented the effective indexing of programming tutorial videos: (i) code in tutorials is typically written on-the-fly, with only parts of the code visible in 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 that is based on: (i) consolidating code across frames, and (ii) statistical language models for applying corrections at different levels, allowing us to make corrections by choosing the most likely token, combination of tokens that form a likely line structure, and combination of lines that lead to a likely code fragment in a particular language. We implemented our approach in a tool called ACE, and used it to extract code from 40 Android video tutorials on YouTube. Our evaluation shows that ACE extracts code with high accuracy, enabling deep indexing of video tutorials.

Categories and Subject Descriptors K.3.2 [*Computers and Education*]: Computer and Information Science Education

Keywords Video Tutorials, Statistical Language Models, Programmer Education

> *Example isn't another way to teach, it is the only way to teach.* – Albert Einstein

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 daily. Video hosting sites such as YouTube host millions of video lectures, while other sites such as SlideShare and Coursera offer educational material on a massive scale. Live streams of programming sessions are also emerging as a valuable educational resource [1], leading to the production of even more programming video content. MacLeod et al. [10] recently investigated the efficiency of on-line videos as a medium for communicating program knowledge among developers and showed its advantages over traditional text-based documentation.

Goal The goal of this work is to enable indexing of programming tutorial videos by accurately extracting the code that appears in them. Accurate extraction makes it possible to *search for a video* by understanding its content as a whole, and to *search inside a video*, for the code of interest. The ability to search tutorial videos has the potential to revolutionize programming education, as well as to allow programmers to find explanations for specific pieces of code during the development process. For example, imagine a world where the integrated development environment (IDE) includes an option to "*find usages with tutorial walkthrough*," where a programmer can find explanations for specific APIs and code pieces. A search can take the programmer directly to the relevant part of the tutorial video, thus effectively creating a "micro-tutorial video" focused around a particular code or API. Indexing tutorial videos is also desirable for referrals to semantically related content and targeting of ads.

Automatic understanding of general video content is a challenging problem [6]. In this paper, we target an easier special case of this problem — where the 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 in each frame. Understanding

what constitutes a "code fragment" requires reasoning 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 much research on automatic indexing of videos, and on extracting text from videos [8], most of it has focused on detecting small amounts of text (e.g., captions, product labels, signs) in a video frame [9]. Many works have also dealt with OCR postprocessing, where statistical language models [17] have been used to improve recognition in a document [21, 24], 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 is based on two key ideas to effectively extract this code: (i) Leveraging cross-frame information to identify code fragments and improve extraction accuracy, and (ii) Using statistical language models [17] 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 accurately extracting code appearing in the video. Our approach uses statistical language models (LM) at several levels: tokens, lines, and fragments, to capture regularities in the code that are later used to improve the accuracy of the 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 can 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.

2. Overview

In this section, 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.

In addition, 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 of how a certain functionality is used. The ability to navigate between different code snippets that may appear in the same video is important for presenting the programmer with a semantic table of contents for a video.

In order to index and search a video, we must be able 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. 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 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 use sampling to choose frames,

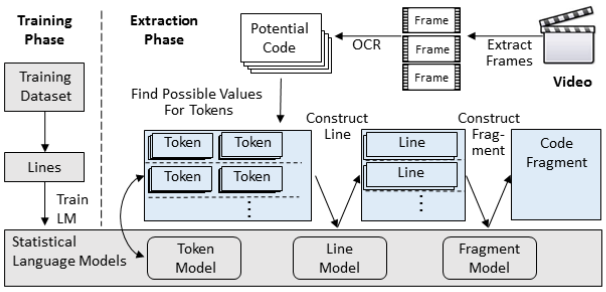

Figure 1: The architecture of ACE.

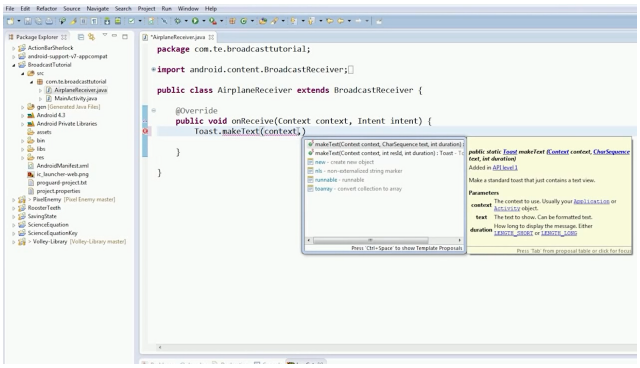

Figure 2: A frame extracted from example tutorial video.

and discard those 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 segmentation of the image into hierarchical containers, and finding the smallest container that seems to cover most of the code in the image. Since the code displayed in the video is usually written on the fly in the IDE, completion suggestion popups may appear and hide part of the editing window with text that is not part of the code. These suggestions should be ignored. Fig. 2 shows a single frame extracted from the example video. In this frame we can identify the main editing window on the right. This is the window from which we would like to extract code. The figure also shows a popup that hides part of the editing window and contains non-code text that will appear in the OCR result if not treated carefully.

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 of snippets as extracted from different frames using OCR are shown in Fig. 3. Our goal is to extract "the best" code snippet that corresponds to the extracted text. We use a simple classifier to identify code written in the target language. The classifier is further explained in Sec. 6.3.

2.2 Extracting Code from Raw Text Frames

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 fragments that can be constructed from the raw text of *all frames*. Towards that end, we leverage two high-level ideas.

Cross-frame Information First, we 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 are used to improve the extracted text in each frame on the basis of 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. 1 as *token model*, *line model*, and *fragment model*. As we explain later, while the *token model* captures token values (as is 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. Furthermore, 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. 3(b)) and should be discarded, or lines where errors can be corrected at the line level (e.g., the first line of Fig. 3(a)), which we explain next.

3. Background and Model

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

3.1 Statistical Language Models

Given a programming language with an alphabet Σ, a language model assigns a probability to each sentence in the language Σ^*. Given a sentence $w_1, \ldots, w_m$ where each $w_i \in \Sigma$ for $1 \leq i \leq m$, we use $P_M(w_1, \ldots, w_m)$ to denote the probability of the sentence as assigned by the language model M. When the model M is clear from context, we drop the subscript and write $P(w_1, \ldots, w_m)$.

```
oackage (om.te.hroad(asttutorial;
'import android.(ontent.BroadcaszeCiner;
public class AirplaneRe(eiver extends Broad(astRe(eiver {
gmxmrreidv
public void intent) {
0 loast.makelext(c0ntext,"Airplane...",loast D
}
}
```

(a)

```
L33 'AuplancRcccrvcrgava Ei-I
: backage com.te.broadcasttutorial;
+inport android.content.BroadcastReceiver;L
public class AirplaneReceiver .tends BroadcastReceiver {
- @Overr%de
A public void onReceive(Context context, Intent intent) {
B Toast.makeText(context,"Airplane...", Toast.LENGIH_$HORT
    ).show()j
}
}
```

(b)

Figure 3: Noisy raw text buffers as extracted from video frames by applying OCR. The string literal "Airplane mode Changed" is shortend for presentation.

```
AirplaneRcccivcnjava toadcastTutorial
AlrplancRcccwcr.Java};
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();
   }
}
```

Figure 4: Code Extracted by ACE.

The probability $P(w_1, \ldots, w_m)$ can be computed using the chain rule, as $P(w_1, \ldots, w_m) = \Pi_{i=1}^{m} P(w_i | w_1, \ldots, w_{i-1})$. In an n-gram model, the probability of a sentence is approximated by only considering a bounded context of length $n-1$. That is, $P(w_1, \ldots, w_m) \approx \Pi_{i=1}^{m} P(w_i | w_{i-(n-1)}, \ldots, w_{i-1})$. For example, in a bigram model, the probability of a sentence is approximated by only considering a bounded context of length 1, that is, $P(w_1, \ldots, w_m) \approx \Pi_{i=1}^{m} P(w_i | w_{i-1})$.

3.2 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.

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).

It is easy to see that 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) according to the lexical rules of the target programming language. We then treat tokens in the raw buffer as having an unknown value, and would like to determine the probability of each token to have a certain value.

For example, Fig. 5 shows the variables in a single symbolic raw buffer with their values as seen in this specific text buffer. Variables are numbered 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 $r_{i,j}^k$ the raw value read at the position from the raw text buffer. For example, the entry $\langle v_{1,9}^1, \textbf{asttutorial} \rangle$ means that the 9th 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 problem 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 represents 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

$\langle v^1_{1,1}$,**oackage**$\rangle\langle v^1_{1,2}$,**(**$\rangle\langle v^1_{1,3}$,**om**$\rangle\langle v^1_{1,4}$,**.**$\rangle\langle v^1_{1,5}$,**te**$\rangle\langle v^1_{1,6}$,**.**$\rangle\langle v^1_{1,7}$,**hroad**$\rangle\langle v^1_{1,8}$,**(**$\rangle\langle v^1_{1,9}$, **asttutorial** $\rangle\langle v^1_{1,10}$,**;**$\rangle$
$\langle v^1_{2,1}$,**'**$\rangle\langle v^1_{2,2}$,**import**$\rangle\langle v^1_{2,3}$,**android**$\rangle\langle v^1_{2,4}$,**.**$\rangle\langle v^1_{2,5}$,**(**$\rangle\langle v^1_{2,6}$,**ontent**$\rangle\langle v^1_{2,7}$,**.**$\rangle\langle v^1_{2,8}$,**BroadcaszeCiner**$\rangle\langle v^1_{2,9}$,**;**$\rangle$
$\langle v^1_{3,1}$,**public**$\rangle\langle v^1_{3,2}$, **class** $\rangle\langle v^1_{3,3}$,**AirplaneRe**$\rangle\langle v^1_{3,4}$,**(**$\rangle\langle v^1_{3,5}$,**eiver**$\rangle\langle v^1_{3,6}$,**extends**$\rangle\langle v^1_{3,7}$,**Broad**$\rangle\langle v^1_{3,8}$,**(**$\rangle\langle v^1_{3,9}$,**astRe**$\rangle\langle v^1_{3,10}$,**(**$\rangle$
$\langle v^1_{3,11}$,**eiver**$\rangle\langle v^1_{3,12}$,**{**$\rangle$
$\langle v^1_{4,1}$,**public**$\rangle\langle v^1_{4,2}$, **class** $\rangle\langle v^1_{4,3}$,**AirplaneReceiver**$\rangle\langle v^1_{4,4}$,**.**$\rangle\langle v^1_{4,5}$,**tends**$\rangle\langle v^1_{4,6}$,**BroadcastReceiver**$\rangle\langle v^1_{4,7}$,**{**$\rangle$
$\langle v^1_{5,1}$,**gmxmrreidv**$\rangle$
$\langle v^1_{6,1}$,**public**$\rangle\langle v^1_{6,2}$,**void**$\rangle\langle v^1_{6,3}$,**intent**$\rangle\langle v^1_{6,4}$,**)**$\rangle\langle v^1_{6,5}$,**{**$\rangle$
$\langle v^1_{7,1}$,**0**$\rangle\langle v^1_{7,2}$,**loast**$\rangle\langle v^1_{7,3}$,**.**$\rangle\langle v^1_{7,4}$,**makelext**$\rangle\langle v^1_{7,5}$,**(**$\rangle\langle v^1_{7,6}$,**c0ntext**$\rangle\langle v^1_{7,7}$,**''**$\rangle\langle v^1_{7,8}$,**Airplane**$\rangle\langle v^1_{7,9}$,**mode**$\rangle\langle v^1_{7,10}$,**Changed**$\rangle$
$\langle v^1_{7,11}$,**''**$\rangle\langle v^1_{7,12}$,**,**$\rangle\langle v^1_{7,13}$,**loast**$\rangle\langle v^1_{7,14}$,**D**$\rangle$
$\langle v^1_{8,1}$,**}**$\rangle$
$\langle v^1_{9,1}$,**}**$\rangle$

Figure 5: The symbolic version (where tokens are turned into variables) of the text buffer from Fig. 3(a).

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, $P_B(v^1_{1,1} = \texttt{package}|\texttt{oackage})$ is very close to 1.

Another approach is to pick token values that are likely for the current frame, that is to maximize $P(v^k_{i,j} = w \mid F_k)$ 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 result in 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 tokes, and possibly 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.

4. From Video to Language Model

In this section, we describe how to train a video-specific language model that can be used to correct the text of a frame based on the text of other (similar) frames. The video-specific language model is combined with a base model that captures regularities in the programming language.

4.1 The Challenge

Consider the first line of the buffer in Fig. 3(a). The raw text of this line is `oackage (om.te.hroad(asttutorial;`. How should we correct the text of this line based on the text of other frames in the video? What is the probability that the first token is actually the Java keyword `package`? What is the probability that the rest of the line is the qualified package name `com.te.broadcasttutorial`?

We can approximate the answers to these questions by constructing a statistical language model that is based on the text buffers of the video. The standard estimation used in an n-gram language models is based on frequency counts. That is,

$$P(w_i|w_{i-(n-1)-1},\ldots,w_{i-1}) = \frac{count(w_{i-(n-1)-1},\ldots,w_{i-1},w_i)}{count(w_{i-(n-1)-1},\ldots,w_{i-1})}.$$

For example, considering a bigram model,

$$P(w_i|w_{i-1}) = \frac{count(w_{i-1},w_i)}{count(w_{i-1})},$$

we can naively treat the buffers of the video as a long sequence of text, and train a statistical model over this text. However, different frames of the video may contain similar text, with different variations. We can leverage this similarity to obtain a more accurate language model. In particular, if we can find *repeated occurrences of similar text*, we can train the model on the best representative of this text. That is, rather than treating the buffers as independent pieces of text, we can use correspondence of positional information to identify corresponding repeated tokens, and use that to train the model.

4.2 Finding (Near) Repeated Occurrences

Consider again the first line of the buffer in Fig. 3(a) `oackage (om.te.hroad(asttutorial;`, and the first token `oackage`. For simplicity, consider a unigram model, in which we only consider the values of individual tokens. Our goal is to find other frames that contain similar text, and

use those to find the most likely value for this token. To find near-repeated occurrences of similar text, we take the following steps:

Step 1 - find corresponding frames: The first step of our algorithm consists of finding frames that contain similar text. Because the code in programming tutorials is mostly written on-the-fly, it could be that certain frames only contain partial code that cannot be parsed, and later frames in the video contain a fuller version of the same code. Further, due to video compression and user interaction (pop-ups etc.), the ability to extract clean text varies between frames. Our first step is to therefore find similar frames so we can leverage repetition between the frames at later stages. For example, the code fragments in Fig. 3 contain similar text.

Step 2 - find corresponding lines: Once we partition the video into sets of similar frames, we identify corresponding lines that repeat across frames. The goal is to identify repetition of lines (or partial lines) and use the aggregate information to improve accuracy. Detecting repetition at the line level is natural and also matches the way OCR extracts the text from the image (which is done via line segmentation). For example, the lines `oackage (om.te.hroad(asttutorial;` and `: backage com.te.broadcasttutorial;` from Fig. 3 contain similar text and would be considered as corresponding lines.

Step 3 - find possible values for tokens: For every token, we collect all its possible values based on the raw texts extracted from all frames.

4.2.1 Finding Corresponding Frames

To leverage repetition across frames, we first need to collect all text buffers that have been extracted from frames that contain similar potential code. Usually, these are neighboring frames, though they might sometimes be frames that are far apart, as is the case for example, when the programmer jumps between several different code fragments, or when the code in some frames is hidden by a popup window.

Finding corresponding frames is actually a clustering problem in which we wish to cluster together frames according to the edit distance. To find corresponding frames, we use the density-based clustering algorithm DBSCAN [4]. Given a set of points and distances between them, DBSCAN groups points that are in proximity to each other and detects the points that are in low density regions as anomalies. DBSCAN requires 3 parameters:

- ϵ is the maximum distance between two points in which they are considered directly reachable from each other. In our case we set the value of ϵ to 0.55.
- `minPts` is the minimum number of points required to form a dense region. In our case we $minPts = 5$, this was set heuristically after probing several values.
- *Distance function*. In our case we use an *edit-distance* metric normalized by the length of the shorter text. We normalize the edit distance by the length of the shorter text since changes to a shorter text are more significant than changes to a longer one.

Finally, we remove clusters that do not contain enough samples. In our experiments we set the threshold at 5% of the total number of frames.

4.2.2 Finding Corresponding Lines

After creating sets of corresponding frames, we wish to leverage the cross frame information. Our goal is to map each token variable $v^k_{i,j}$ to all of its corresponding token variables $v^{k'}_{i',j'}$ (for $k' \neq k$). Recall that 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 find corresponding lines. Corresponding lines are lines from different frames that seem to be occurrences of the same line of code. That is, for each pair of k, i,we wish to find all of its corresponding pairs k', i'. 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 normalized edit-distance function. The next text buffer to align is the one with the lower edit distance to one of the already aligned text buffers.

In order to detect anomalies within the set of aligned lines (lines that were similar enough to be added to the alignment but are far apart from the majority of the lines) we use the DBSCAN algorithm. We apply DBSCAN on each set of corresponding lines, with lower $\epsilon = 0.25$, and choose the largest cluster as the set of corresponding lines. If the largest cluster does not contain at least half of the aligned lines, we do not consider it as a set of corresponding lines since it is more likely that these lines are the result of noise.

4.2.3 Finding Possible Values for a Token

The first step in finding possible token values "as seen in the video" is to find corresponding tokens. Given a set of corresponding lines, our goal is to find for each token variable all of its corresponding token variables. Some of the lines may be partial because of popups or because they have been captured while the programmer was writing them. But aside from partial lines, all corresponding lines in a set are similar to one another.

Because OCR does not duplicate tokens nor change the order of tokens within a line, we can split and then align corresponding lines to find corresponding tokens. We split each line to tokens using a modified version of the programming language lexer, modified so that 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, we align all corresponding lines. Typically, we deal with dozens of lines in each group of corresponding lines, and 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 two lines that are most similar. Then we align the rest, line by line, to those that are aligned. The next line to be aligned is the one that is the most similar to any of those that have already been aligned. Finally, we remove sets that do not contain enough tokens since these sets are probably garbage. This technique allows us to compute the frequency counts required for building a language model.

5. Extracting Code from Video

In this section, we describe how we use the statistical language models to extract code fragments from the video. We use several generic base language models that capture regularities in the programming language, as well as the specific video language model constructed in Section 4.

5.1 Base Language Model

In addition to token values extracted from the video frames, we also consider token values from a statistical language model of the target language. The idea is that even if all values for a token in the video are noisy (incorrect), we can use similarity to other common tokens (in the language) in order to correct it. This is particularly useful for keywords, common type names, and common package names. Fig. 3 shows two raw text buffers as extracted from the video. In snippet *(a)* we can see the raw value **oackage**, while in snippet *(b)* the extracted raw value to the corresponding token is **backage**. We create a set of all possible values as seen in the video, including **oackage** and **backage**. Had these been the only values in the video, the probability of this token having the (correct) value **package** would have been zero (or close to zero after smoothing). Although this is an extreme (and not likely) example, it demonstrates the value of combining our video-specific language model with a base language model. We use a base language for Java, which we train on over 1-million code snippets. In the unigram base model, the token **package** is very common.

5.2 Line Structure Model

In addition to base language models for token values, we use a language model that captures common *line structures*. This model assigns probabilities to line structures based on token types. In a sense, this model captures frequent line structures permitted by the grammar of the language. We use $P_{LS}(l)$ to denote the probability assigned by the line-structure model to a given line structure l.

For example, our line structure model assigns a higher probability to the line structure `IMPORT ID DOT ID DOT ID SEMI` than to the line structure `SUPER LPAREN ID RPAREN SEMI`. That is, a line that performs an `import` from a package is more common than a line that calls the constructor of a superclass with a single parameter.

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 or adding certain tokens. 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 Section 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 a 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 below. For each line we keep the set of most likely line structures.

Forcing Structure on a Sequence of Token Variables Given a line structure and a sequence of token variables with their possible values, our goal is to assign a possible value to the token variables, while possibly 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 number of matched token variables. Token variables that did not match any of the symbols are removed, while symbols that did not match any of the token variables are added.

For obvious reasons, if the added token variables are not assigned 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, others — identifiers for example— can still be assigned more than one option.

Assignment of values to token variables is based on the video-specific language model P_V, and the base model for the programming language. We assign values to tokens in a way that maximizes the following equation for a given line l containing words $w_1, \ldots, w_m$:

$$P_{LS}(l)P_V(w_1, \ldots, w_m) \approx P_{LS}(l) \prod_{i=1}^{m} P_V(w_i \mid w_{i-1}).$$

Example Fig. 6 shows possible values for a sequence of tokens $t1, \ldots, t7$. We can force these tokens into two possible line structures from the statistical model:

$\underline{t1}$	$\underline{t2}$	$\underline{t3}$	$\underline{t4}$	$\underline{t5}$	$\underline{t6}$	$\underline{t7}$
oackage	om	.	te	.	asttutorial	;
backage	com	,		,	broadcasttutorial	
package						

Figure 6: Constructing a Line from Possible Values for Tokens

- the line structure `PACKAGE ID DOT ID DOT ID SEMI` leading to **package** $t2$**.**$t4$**.**$t6$**;** with some choice for the identifiers;
- the line structure `ID LPAREN ID COMMA ID COMMA ID RPAREN SEMI` leading to $t1$**(**$t2$**,** $t4$**,** $t6$**);** with some choice for the identifiers.

In this example the first structure requires neither removal nor addition of tokens and therefore this structure will be the one we choose to force. The assignment for tokens $t1, t3, t5$ and $t7$ was chosen based on the structure constraints, but we still have to choose the most likely assignment for the rest of the tokens in the line.

The assignment for $t3$ is chosen to be the only option **te**, but $t2$ and $t6$ both have two options to choose from.

We first try to assign values based on a statistical language model of token sequences. Using this model assigns $t2$ with the value **com** since it is more likely to be seen in the programming language. Finally, we assign the value **broadcasttutorial** to $t6$ because it is the value that occurs most frequently in the video.

It is important to note that the *line model* captures the syntactic structures of a line of code. The decision is line based and not statement based for two reasons: (i) OCR works at the line level, and garbage lines can appear between two lines that constitute the same statement, and (ii) the displayed code may be partial and therefore statements may be partial as well.

5.3 Fragment Level Model

The result of forcing line structure on each line can be more than one possible assignment. We choose the most likely line structure for each line based on a bigram model over line structures. The result of applying ACE on the example video is shown in Fig. 4. Note that this code fragment differs significantly from the examples of text extracted by OCR as shown in Fig. 3.

Forcing line structure on a sequence of token variables, results in 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 number of modifications.

Since the number of changes is the same, a naive method would be to choose the assignment according to its probability in the line structure model. This naive method does not consider the context. We would like to choose the most likely assignment for each line on the basis of not only the line structure but also of other lines in the same fragment. To assign to each line its most likely structure in context, we use a language model over line structures. We provide details about the training phase in Section 6. We assign structures to lines in a way that maximizes the overall probability.

6. Implementation

We implemented our approach in a tool called ACE (Automatic Code Extraction). 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 that teach how to write code for the Android platform. ACE is implemented both in Java and Python and relies on OpenCV [3] for frame extraction and image processing, on the Tesseract OCR library [18] for text extraction, on scikit-learn [14] for clustering, and on ANTLR [13] 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. These videos often show more than 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 distinguish 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 uniform sampling. In the future, we plan to employ more advanced segmentation 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 [3] 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 times impossible due to limitations of image processing techniques. We therefore find the main editing window in one of the samples frames 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 further described in Sec. 6.3), 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 Identifying Instances of the Target Language

After extracting text buffers from a video, ACE must identify those that contain potential code written in the target language. 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 whether 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 use 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 into a sequence of tokens using the programming language lexer. We process the sequences 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. We thus obtain a set of sequence of leaves per 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 class, interface, and constant names while variables and method 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 a 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 one.

7. Evaluation

In this section, 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 they are very popular on video hosting sites, and the number of such available videos is huge. Even with HD quality, direct text extraction is insufficient for code indexing (as shown in Fig. 3). It is important to note that our techniques also apply to videos of lower (and higher) quality. For SD videos, the approach is typically accurate enough to pick up keywords, but not for extracting complete code fragments. Our techniques for consolidating code across frames, handling code that is written on the fly, or handling obstructions due to pop-ups 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.

Code in a Video A programming tutorial video may contain multiple code snippets. Moreover, 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? This 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; 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.

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 recall and precision of our extraction by comparing tokens at the leaf level.

Video Sampling Method We uniformly sample 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. This guarantees a minimal number of samples.

7.2 Results

Table 1 shows the programming tutorial videos we used as benchmarks, the number of code snippets extracted manually from the video, and the number of snippets extracted by ACE. 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 (where we failed to merge two snippets despite 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 *we often count a single snippet where different frames show different parts*

Table 1: Programming tutorial videos and number of extracted snippets using uniform sampling method. `#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.

Name	Duration (M:S)	S	#Fr	D	E
Android Eclipse.learn	1:18	1	170	1	0
30 -Color of TextView	3:56	1	201	1	0
Android Tutorial 23	4:06	1	173	1	0
Android Button	4:46	1	118	1	0
How To: Android	4:55	1	109	1	0
How to stream video	5:22	2	133	1	0
Android Tutorials 62	6:15	3	68	2	0
Android App. Dev.	6:59	1	168	1	1
Android Tutorial 64	7:00	1	54	1	0
Android Tutorial 19	7:28	3	143	3	0
Android Tutorial 29	8:03	1	104	1	0
Android Tutorial 61	8:09	1	50	1	0
Android Tutorial 63	8:45	2	84	2	0
Android Chronometer	9:05	1	164	1	3
How to read file	9:16	1	143	1	1
Android Create Menu	9:28	5	105	1	0
Android Tutorial 7	9:47	2	137	2	0
Android Tutorial 18	9:53	3	92	3	0
Fragment Tutorial	9:54	4	109	2	0
Android Tutorials	10:30	1	130	1	0
Android Tutorial 43	11:19	1	128	1	0
Android Tutorial 38	11:19	2	141	2	0
Android Tutorial 55	11:46	1	120	1	0
Android Tutorial 17	11:55	5	129	2	0
Delete Selected	11:56	1	163	1	1
Android Tutorial 13	12:02	2	154	1	0
Android Web View	12:03	1	98	1	0
Basic ListView Demo	12:54	1	135	1	0
Android Tutorial 48	13:16	1	99	1	0
Android Alert Dialogs	14:07	1	109	1	1
Android Tutorial 22	14:20	2	181	1	0
App in 15 minutes	14:39	1	80	1	0
Android Tutorial 5	15:06	1	124	1	1
File Read	15:19	1	161	1	0
41. Drag and Drop	15:28	1	156	1	1
Android Tutorial 36	17:03	3	134	2	0
Android Tutorial 21	18:00	3	151	2	0
Android Tutorial 68	21:35	2	174	2	0
Android Tutorial 72	25:14	2	260	2	1
Android Pure Java	30:44	1	107	1	0

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 written on the fly. This means that different frames almost always show different parts of the code (with some overlaps).

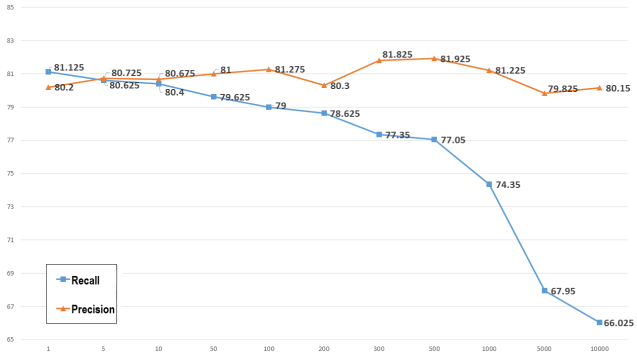

Figure 7: Recall and precision for varying thresholds.

In some videos, such as `Android Create Menu`, 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 `Android 4.0 Tutorials - 22` (for short `#22`), there are two snippets, but one of them is only used for reference, where the author jumps between the main code being written and the reference snippet, which is only shown for short periods. With the current parameters only the main snippet is extracted.

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 (with the current extraction parameters).

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 snippets, resulting in a total of 3 extra snippets.

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

Recall and Precision We measure the recall 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. We evaluate the snippets in the video that ACE detected. Recall stands for the percent of correct extracted tokens out of the ground truth tokens. Precision stands for the percent of correct extracted tokens out of the extracted tokens. Tokens are correctly extracted tokens if both the grammar symbol and value are correct. 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):

```
PUBLIC VOID <ID_l,"onReceive"> LPAREN <ID_u,"Context">
<ID_l,"context"> COMMA <ID_u,"Intent"> <ID_l,"intent">
```

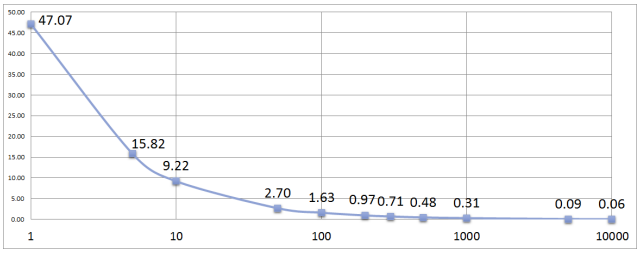

Figure 8: Percentage of line structures considered valid as a function of number of occurrences in the model.

```
RPAREN LCBR
```

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
```

then, we will report that there are 4 missing tokens, and 2 redundant tokens. Hence, the recall for this line is 7/11, and the precision is 7/9.

Fig. 7 shows the recall and precision with different threshold values for the number of occurrences required for a line structure to be considered valid in the language model. 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 recall of 66.025 and precision of 80.15. The recall is low since we try to force wrong line structures on valid lines, causing tokens to receive the wrong grammar symbol. However, if we consider any line structure that appears more than once as valid, we observe a recall of 81.125 and precision of 80.2. The recall is high because we consider more line structures as being valid. The precision remains the same because, as the threshold increases, so does the number of lines that do not match any line structure. This in turn causes both valid lines and garbage lines to be considered invalid.

Fig. 8 shows the percentage of different line structures which are considered valid as a function of the required number of occurrences in the model. 47% of the lines appear more than once. Some line structures are very common and occur tens of thousands of times. Fig. 9 shows the ten most popular line structures in the model. Since programming for the Android platform involves the plentiful use of libraries, second and fourth place in the list are occupied by `import` statements. Furthermore, the third, sixth and tenth place are occupied by inheritance related statements, since Java is an object oriented language that involves the massive use of inheritance.

Extraction Times We measure the time required for processing each video, which includes the time it takes to

- extract raw text from the video (this includes video processing, image processing, and OCR);
- find corresponding frames;
- find corresponding lines;
- find possible values for tokens;
- force line structure;
- carry out 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, although in 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 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.

Indexing ACE makes it possible to index programming tutorial videos. For each code snippet in a video, ACE detects the frames that present the code snippet, even if it doesn't appear as a whole. As our ground truth we watched 3 videos and manually extracted the time intervals in which each code snippet appears. Then we compared the frames ACE detected for each code snippet with the manually extracted intervals. The samples ACE took are the elements for this test. The samples that belong to a code snippet interval are the relevant samples. Relevant samples recognized byACE as belonging to the correct code snippet are considered true positives, relevant samples not correctly recognized are considered false negatives and non-relevant samples detected by ACE as belonging to a code snippet interval are considered false positives. For example, pop-ups can cause a false-negative result as ACE might fail to recognize a frame with pop-ups as being part of a code snippet interval. We evaluated ACE over 3 programming tutorial videos, each of which presents a different number of code snippets. Fig. 2 shows the precision and recall results for each video. Recall equals 1 for each video, meaning that every sample is recognized by ACE and matched to the relevant code snippet. Precision is also high. We thus conclude that ACE indexing is accurate.

Rank	Structure	Example
1	RBRACE	}
2	IMPORT ID_l DOT ID_l DOT ID_u SEMI	import android.app.Activity;
3	AT ID_u	@Override
4	IMPORT ID_l DOT ID_l DOT ID_u DOT ID_u SEMI	import android.view.View.OnClickListener;
5	PRIVATE ID_u ID_u SEMI	private UUID Id;
6	PUBLIC CLASS ID_u EXTENDS ID_u LBRACE	public class Configue extends Activity {
7	PACKAGE ID_l DOT ID_l DOT ID_l SEMI	package com.imps.activities;
8	PUBLIC ID_u ID_l LPAREN RPAREN LBRACE	public UUID getId(){
9	PACKAGE ID_l DOT ID_l DOT ID_l DOT ID_l SEMI	package android.speech.tts.location;
10	SUPER DOT ID_l LPAREN ID_l RPAREN SEMI	super.setAltitude(altitude);

Figure 9: Top ten line structures.

Name	#Snippets	#RS	#TP	Precision	Recall
Android Tutorial 55	1	115	97	0.84	1
Android Tutorial 68	2	176	163	0.93	1
Android Tutorial 19	3	149	143	0.96	1

Table 2: Recall and precision of ACE indexing. #RS is the number of relevant samples and #TP is the number of true positive samples.

Standalone OCR indexing inefficiency We evaluated whether videos could be indexed by simply applying OCR on sampled frames. Though code snippets contain dozens of unique tokens, applying OCR on sampled frames results in *thousands* of unique tokens. This proves that stand-alone OCR indexing is inefficient, with less than 1% precision.

8. Related Work

In this section, we survey closely related work.

OCR Post-Processing There has been a lot of work on OCR post-processing. OCR technology itself is not precise enough to produce quality results when extracting code from videos [9], especially in 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 [21] post-process OCR results using a correction system that is 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. [24] post-process OCR results using n-grams and latent semantic analysis language models to obtain both syntactic and semantic information. Taghva et al. [19] 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 [11] index presentation videos using the text in the displayed slides. They extract the text directly from the video and use textual changes to segment the video into semantic shots. They apply image processing techniques to deal with low image quality and do not use OCR post-processing.

Yang et al. [23] index lecture videos by first segmenting them 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 most correct words. If two or more extracted texts have the same number of correct words, they choose the one with the lower word count, and if the texts contain 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 [23]) to perform error correction.

Hua et al. [7] use the concept of multiple frame integration to improve text recognition. They join blocks from multiple frames to create a clearer frame, and send it to the OCR engine for recognition. Our approach also uses the concept of multiple frame integration to improve extraction. While Hua et al. improve text extraction by creating a clearer frame as OCR input, our approach improves code extractions by merging the OCR results. Our approach further enables the merging of OCR results that contain different parts of the same code snippet.

Error Correction in Parsers Our approach is also related to automatic error correction in parsing [2, 20]. 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 a code completion engine for Java using a trigram model that suggests a next token.

Raychev et al. [16] 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. [22] show that many 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 for correcting code from programming tutorial videos.

Nguyen et al. [12] 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 breaks into lines is critical in the OCR setting, where segmentation is line-based and entire lines may end up being garbage due to noise.

Indexing Tutorial Videos Recently, Ponzanelli et al. [15] developed CodeTube, a Web-based recommender that supports video tutorials for software engineering. They extract text from tutorial videos using OCR and identify frames as belonging to the same code snippet by using an island parser to locate a common Java construct in the frames. When the parser fails due to noise, they use edit distance and image similarity techniques. Our experience indicates that OCR over frames of programming tutorial videos may produce some useful matches, but overall yields very low precision. Therefore, in contrast to directly applying OCR (that can also be used for indexing), our approach performs *accurate* extraction by using cross-frame information and statistical language models.

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, others, especially low quality videos, make it difficult to find contours. Our method relies on the fact that the number of potential code tokens in a single line is significantly larger than the number 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; applying OCR on these areas will yield disproportionately noisy results as compared to applying OCR only on the main editing window.

Unique fonts: While our method can handle splitting token errors and character modifications within a token, it cannot handle cases in which two or more identifier tokens are consistently merged into one. Since our method 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 merged. 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 a 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, the OCR tool should be trained 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 produces more noise than our method can handle.

Glances at code: When the author jumps from one code fragment to another without significant pauses, we cannot differentiate between frames that contain code from extremely noisy frames. Our method relies on the repetitiveness of code in a tutorial video and therefore treats singular code frames as noise.

10. Conclusion

We believe that video demonstrations and lectures are a rapidly growing and important driver of programming education. In fact, the ability to extract and index this information could already provide tremendous value to programmers by directing them to relevant and timely sources. This work is a first modest step in the attempt to better leverage this wealth of information.

We presented an approach for extracting code from videos and indexing programming tutorial videos. We show that *precise extraction* requires more than direct application of OCR, and can be made possible by integrating techniques from image processing, parsing, and statistical language models (trained on "big code").

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 form 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.

Acknowledgments

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7) under grant agreement no. 615688 -ERC-COG-PRIME.

References

[1] Watch people code, 2016. URL http://www.watchpeoplecode.com/. [Online; accessed 20-August-2016].

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[3] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, Inc., 2008.

[4] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

[5] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *34th International Conference on Software Engineering*, ICSE. IEEE, 2012.

[6] W. Hu, N. Xie, L. Li, X. Zeng, and S. Maybank. A survey on visual content-based video indexing and retrieval. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 2011.

[7] X.-S. Hua, P. Yin, and H.-J. Zhang. Efficient video text recognition using multiple frame integration. In *Procceding on the International Conference on Image Processing*. IEEE, 2002.

[8] R. Lienhart and A. Wernicke. Localizing and segmenting text in images and videos. *IEEE Transactions on Circuits and Systems for Video Technology*, 2002.

[9] T. Lu, S. Palaiahnakote, C. L. Tan, and W. Liu. *Video Text Detection*. Advances in Computer Vision and Pattern Recognition. Springer, 2014.

[10] L. MacLeod, M.-A. Storey, and A. Bergen. Code, camera, action: how software developers document and share program knowledge using YouTube. In *Proceedings of the IEEE 23rd International Conference on Program Comprehension*, 2015.

[11] M. Merler and J. R. Kender. Semantic keyword extraction via adaptive text binarization of unstructured unsourced video. In *16th IEEE International Conference on Image Processing*, ICIP, 2009.

[12] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. A statistical semantic language model for source code. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE. ACM, 2013.

[13] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

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

[15] L. Ponzanelli, G. Bavota, A. Mocci, M. Di Penta, R. Oliveto, B. Russo, S. Haiduc, and M. Lanza. Codetube: extracting relevant fragments from software development video tutorials. In *Proceedings of the 38th International Conference on Software Engineering Companion*. ACM, 2016.

[16] V. Raychev, M. Vechev, and E. Yahav. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2014.

[17] R. Rosenfeld. Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*, 2000.

[18] R. Smith. An overview of the tesseract ocr engine. In *Proc. Ninth Int. Conference on Document Analysis and Recognition (ICDAR)*, 2007.

[19] K. Taghva, T. Nartker, A. Condit, and J. Borsack. Automatic removal of "garbage strings" in OCR text: An implementation. In *The 5th World Multi-Conference on Systemics, Cybernetics and Informatics*, 2001.

[20] R. Teitelbaum. Context-free error analysis by evaluation of algebraic power series. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC, 1973.

[21] X. Tong and D. A. Evans. A statistical approach to automatic OCR error correction in context. In *Proceedings of the Fourth Workshop on Very Large Corpora*, 1996.

[22] Z. Tu, Z. Su, and P. Devanbu. On the localness of software. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, 2014.

[23] H. Yang, M. Siebert, P. Luhne, H. Sack, and C. Meinel. Lecture video indexing and analysis using video OCR technology. In *Seventh International Conference on Signal-Image Technology and Internet-Based Systems*, SITIS. IEEE, 2011.

[24] L. Zhuang, T. Bao, X. Zhu, C. Wang, and S. Naoi. A chinese OCR spelling check approach based on statistical language models. In *IEEE International Conference on Systems, Man and Cybernetics*, 2004.