

Turnitoff: Identifying and Fixing a Hole in Current Plagiarism Detection Software

James Heather

Abstract

In recent times, *plagiarism detection software* has become popular in universities and colleges, in an attempt to stem the tide of plagiarized student coursework. Such software attempts to detect any copied material and identify its source. The most popular such software is Turnitin, a commercial system used by thousands of institutions in more than one hundred countries.

Here we show how to fix a loophole in Turnitin's current plagiarism detection process. We demonstrate that, in its current incarnation, one can easily create a document that passes the plagiarism check regardless of how much copied material it contains; we then show how to improve the system to avoid such attacks.

1 Introduction

Plagiarism is not a modern phenomenon. Controversies surrounding plagiarism stretch back at least as far as Newton and Leibniz and the origin of calculus; and Shakespeare has been accused of copying from Marlowe, Heywood, Chaucer and others. The temptation to find one's fame and success by standing, unconfessed, on the shoulders of giants is not a new one.

The battleground has changed markedly in recent times, however, owing to the growth of the Internet. One can find sources to copy with a quick web search, where before one would have had to search more diligently through books and journals; one can appropriate the text with a quick copy-and-paste operation, where before one would have had to type (or hand-write) the text out. But detection and conviction are also markedly easier: as any academic knows, a web search on a suspect phrase will often find the source of the plagiarized part of the essay.

As a result, recent years have seen the rise of *plagiarism detection software*, specifically designed to take in submissions of student (or other) work and check for copied content. These are designed to allow lecturers to discover with ease when students have copied from each other or from other sources.

A large number of universities and colleges now have some such system in place. Often the plagiarism detection software is linked to the system students are required to use for submitting their work, so that any work submitted electronically is automatically checked for plagiarism, and some sort of measure of similarity to other work in the system's database is returned to the operator. Submissions with a high score can then be checked in more detail.

It should be acknowledged from the start that plagiarism is tricky to define and sometimes difficult to analyze. It is increasingly recognized that the whole notion of plagiarism is a culturally dependent one (Gu and Brooks 2008; Currie 1998), and that the boundaries between plagiarism and the normal learning process are not always clear cut (Pennycook 1996). Many of our assumptions about student behaviour are based on Western culture, and do not sufficiently take into account the difficulties overseas students may face when studying in Western institutions (Hayes and Introna 2005). Plagiarism, therefore, always needs to be dealt with carefully and sensitively, and not all cases treated in the same manner and with the same severity.

For the purposes of this paper, however, we will be considering plagiarism of the most deceptive kind: intentional plagiarism, with a conscious effort to cover one's tracks. It is our contention that avoiding detection is rather easier than commonly believed; and that unless improvements are made to current plagiarism detection technology, we are in danger of identifying and punishing the lower level cases while allowing the most serious offences to slip through undetected.

1.1 Plagiarism detection software

We begin by briefly considering several available plagiarism detection systems, and then looking at how such systems operate.

1.1.1 Current systems

There are several plagiarism detection systems available, with varying functionality.

The most comprehensive survey of existing systems is to be found in a study (Scaife 2007) of eleven plagiarism detection systems, commissioned by the UK's Joint Information Systems Committee Plagiarism Advisory Service (JISC-PAS), conducted for JISC-PAS by the NCC Group. The study highlighted four complete plagiarism detection suites: Ephorus Suite, written by Ephorus¹; MyDropBox/SafeAssign, developed by Blackboard²; Turnitin, created by iParadigms, LLC³; and Urkund, by PrioInfo AB⁴.

All four of these suites search not only web sites, but also all previously received submissions. When a student submits an essay, it is checked against previous submissions as well as other potential sources; this helps to identify students who copy from each other on the same piece of work. Ephorus, MyDropBox and Turnitin all have modules that support integration with various popular course management systems so that all student submissions can be analyzed for plagiarism automatically. (Urkund operates in a rather different fashion: it is an email-based service.) All four also allow submission in various formats, including Microsoft Word documents or PDF files; and since anything that can be printed can also be turned into a PDF document, this effectively encompasses all electronic documents.

¹<http://www.ephorus.com/home>

²<http://www.safeassign.com/>

³<http://www.turnitin.com/>

⁴<http://www.urkund.com/>

Despite these similarities, the JISC-PAS study unequivocally recommended Turnitin as their system of choice. The result of the study was that Turnitin came top in every category that the NCC investigated. In consequence, JISC-PAS now maintains on its web site⁵ that

Turnitin is the global leader in electronic plagiarism detection, is a tried and trusted system and over 80% of UK universities have adopted it.

Turnitin is equally strong in the USA, where it is employed at, to name but a few institutions, Georgetown University, Miami Dade University, Auburn, and UCLA; and in over one hundred other countries, including Canada, Australia and Singapore. Turnitin claims on its web site that it has so far checked over 75 million submissions for plagiarism.

In view of Turnitin's dominance in the marketplace, the rest of this paper will focus largely on avoiding detection in Turnitin; however, the various methods we shall employ are all general enough that they will work on any system that operates in the same fashion as Turnitin.

1.1.2 How such systems work

Plagiarism detection software typically operates in three distinct stages:

1. extract the text from the document (usually a PDF or Microsoft Word document);
2. search for the text online and (possibly) in a customized database;
3. quantify the extent to which extracted text could be found in other sources, and display results.

It is perhaps the second stage that is the most complex and is usually considered the most difficult. Searches have to be able to cope with text where the occasional word has been changed, so it is not enough simply to take whole sentences and attempt to find them elsewhere. Performing such 'fuzzy' searches in a reasonable timeframe, with a large corpus of searchable material, is not easy.

However, it is the first step—extraction of the text—that is most vulnerable to attack.

Section 2 will consider how one can submit a plagiarized document without being detected, by producing the document as a PDF that looks and prints as normal but that inhibits extraction of the text. We then argue in Section 3 that with the current approach to text extraction, there is a very high chance that such a rogue submission will pass undetected. In Section 4, we show how to harden the text extraction phase to defeat such attacks. We sum up and draw conclusions in Section 5.

2 Avoiding detection

Plagiarism detection software works by extracting the text from the document being checked, and comparing the text with other sources (from a customized database, or from the Internet, or both).

⁵<http://www.jiscpas.ac.uk/turnitinuk.php>

Ligatured (1 glyph)	Unligatured (3 glyphs)
ffi	ffi

Figure 1: Ligatures

Avoiding detection involves attacking this first step: if we can stop the text from being properly extracted from the document, without affecting how the document looks and prints, then the software will not be able to identify any plagiarized material in the document.

We demonstrate in this section how one may modify a document to be submitted for a plagiarism check, so that any plagiarism in the document will go undetected. All three of the methods presented here are trivial to implement, and require no special software beyond what is freely available for download from the Internet.

In all three methods, what is ultimately produced for submission is a PDF. The first method involves creating the PDF using the pdfL^AT_EX typesetting system; the second and third allow it to be produced using any word processing or typesetting system, including Microsoft Word.

2.1 Modifying the character map

Text in a PDF is typically stored as a sequence of *Character Identifiers*, or *CIDs*. These CIDs are treated as pointers to the character glyphs (lines and curves describing the shapes of the characters) that make up the font. Rendering the document to screen or printer involves looking the CID up in the font table and placing the glyph on the page at the appropriate point.

These CIDs do not in every case denote individual letters of the alphabet. Sometimes fonts employ *ligatures* to improve the appearance of printed text. In the word ‘stuffing’, for instance, the character sequence ‘ffi’ will in many fonts rendered as a single symbol (Figure 1). Fonts also do not all have their glyphs stored in the same order. As a result, it is difficult to extract the text from just the CIDs and glyphs.

For this reason, a PDF also typically contains a further mapping—a *Character Map*, or *CMap*—to enable reconstruction of the text. This is a mapping from CIDs to *character codes* (or sequences of character codes in the case of ligatures); and these character codes do unambiguously represent individual characters of the text.

One simple approach to avoiding extraction of the text is therefore to modify the CMap. Swapping the entries that point to the character codes for ‘x’ and for ‘y’ will result in a PDF that looks and prints as before (because the CIDs and glyphs have not been altered), but that has instances of ‘x’ and ‘y’ swapped in the extracted text. Applying a random reassignment of the entries will produce completely garbled extracted text throughout the document.

Documents that are typeset in pdfL^AT_EX (such as this paper) can easily have their CMap altered. A standard L^AT_EX 2_ε package, `cmap`, allows one to place a custom CMap into a document. In the main file, one needs only

```
\usepackage[resetfonts]{cmap}
```

to ask pdfL^AT_EX to insert the CMap corresponding to the font being used. The default L^AT_EX font uses the OT1 encoding, so it will search for a file called `ot1.cmap`, and use that if present. To create a CMap suited to our purposes, we can make a copy of the T1 encoding (file `t1.cmap`, which comes included with the `cmap` package) as `ot1.cmap`, and place it in the same directory as the document we are editing⁶. We need to make one small change to alter the CMap and confuse anything that attempts to extract the text from the PDF: open the new `ot1.cmap` file in a text editor, and find the line that reads

```
<61> <7E> <0061>
```

This is to be interpreted as an instruction to map character codes 61 to 7E (hexadecimal) to Unicode characters starting at `<0061>`, which represents a lower case ‘a’. If we change this to

```
<61> <7E> <0062>
```

the effect is to map to Unicode characters starting at ‘b’. Now when we generate the PDF, the mangled CMap will be included, and any attempt to extract the text will be defeated. Whenever an ‘a’ appears in the document, it will be seen as a ‘b’; a ‘b’ will be seen as a ‘c’; and so on. This can be easily verified by opening up the PDF with a suitable document viewer and attempting to search for words and phrases in the document. By including one standard package, and making a one-character edit to a single file, we have invisibly altered all of the lower case letters in the document. Another one-character edit, if desired, would deal similarly with the upper case characters.

Passing such a document through Turnitin results in a very low score, because it sees the text as nonsense. Turnitin’s report on a sample PDF built in this way is shown in Figure 2. Note the score of 0%, and the garbled extracted text.

The modified `ot1.cmap` and sample PDF, along with other supporting files, are available online⁷.

2.2 Rearranging the glyphs in the font itself

Another option is to alter or rearrange the glyphs themselves, so that the mapping from character codes to glyphs no longer corresponds to the normal alphabet.

What is visible to the plagiarism detection software is the stored text before the translation into glyphs occurs. Figure 3 shows the glyphs associated with the letters ‘e’ and ‘x’ in a typical font.

The human reader sees the glyphs rather than the text itself. The equivalence of the two relies on there being a sensible mapping of characters to glyphs inside the font; that is to say, an ‘x’ in the text comes out looking like an ‘x’ only because the font correctly maps the letter ‘x’ to a glyph that has a shape that looks like ‘x’. If the font mapped an ‘x’ to a shape that looked like an ‘e’, the

⁶We here use `t1.cmap` as the starting point rather than `ot1.cmap` because some older distributions do not contain the latter; but either will work fine.

⁷<http://www.computing.surrey.ac.uk/personal/st/J.Heather/plagiarism/>

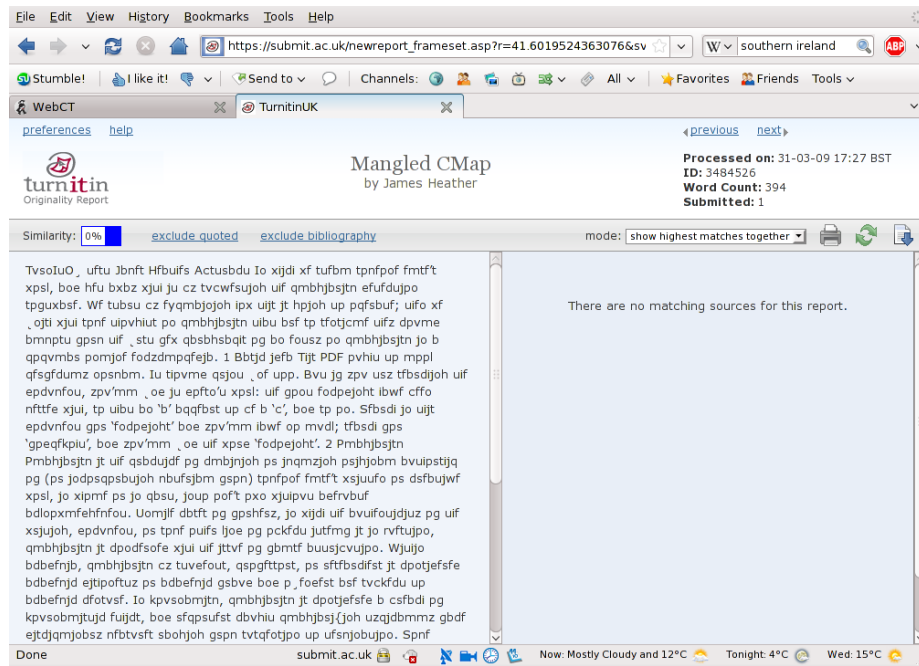


Figure 2: Submitting a PDF with a mangled CMap to Turnitin



Character	e	x
Glyph		

Figure 3: Glyphs in an unmodified font



Character	e	x
Glyph		

Figure 4: Glyphs in a modified font

effect would be that the text inside the PDF would still contain an ‘x’, but to a human, it would appear to be a letter ‘e’.

This opens up an interesting possibility. If we modify a font so that it contains the same glyphs as before but with some of them swapped round, then the link between the text and its printed representation will be broken.

Figure 4 shows a font in which the glyphs for ‘e’ and ‘x’ have been swapped. Opening up Microsoft Word or OpenOffice Writer, selecting this font, and typing in ‘xetra’ results in what appears to be ‘*extra*’ being typeset in the document. This is how it will appear on the screen, and this is how it will print; but internally it represents the (non-)word ‘xetra’. The spell checker will treat it as such; a search-and-replace will do the same; and, crucially for our purposes, so will a plagiarism detection system like Turnitin.

To produce a document that clears Turnitin by this mechanism, one has to perform the following steps.

1. Create a modified version of a standard font with some letters switched round. This can be done quickly and easily with a program like Font-Forge⁸, which is free software, available for Linux, Windows and Mac. This needs to be done only once; subsequently, the same font can be reused for any number of documents.
2. Type the document in the usual way, initially using the standard font. This can be done in any word processing software, including Microsoft Word and OpenOffice Writer.
3. Identify the parts of the document to be masked from the plagiarism detection software. Switch these parts to using the modified font.
4. Use search-and-replace on those parts of the document to switch letters around so that the document looks and prints as normal.
5. Export the document as a PDF, and submit it.

The last step is necessary because a typical word processor document (for example, a Microsoft Word document) does not contain the font itself. If the document were to be opened on a machine that did not have the modified font installed, it would be rendered using a different font, and the alterations would be visible. A PDF, by contrast, can have the fonts embedded inside, and so would look the same regardless of what fonts are installed on the machine being used to view or print the document.

Figure 5 shows Turnitin’s report from a sample PDF created with such a modified font. Note again the score of 0%, and the nonsense text.

⁸<http://fontforge.sourceforge.net/>



Figure 5: Submitting a PDF with rearranged glyphs to Turnitin

The modified font and sample PDF are available online at the URL given in Section 2.1.

2.3 Converting text to Bézier curves

A third approach is to replace all text in the document with the Bézier curves that represent its shape. For this, we need a PDF, but the source of the PDF is of no importance: it can be produced in any way we wish, including using Microsoft Word or any other word processing package. After typing (or, perhaps, copying and pasting!) the document, we convert it to a PDF; we are now ready to process the PDF so that it passes the plagiarism check.

The fact that the text is stored in the document is crucial for how plagiarism detection systems work: they need to recover the text in order to compare it with other documents. If there is no text, then the plagiarism detection cannot function.

However, a PDF can also represent arbitrary Bézier curves, drawn anywhere on the page; this is essential for allowing diagrams and graphs to be displayed. We can take advantage of this by taking each character on the page and replacing it with the Bézier curve that defines the character glyph. This does not visibly alter anything—after all, we are simply performing the same rendering process that the PDF viewer would perform. The effect on anything that attempts to extract text from the document is, however, to make all the text invisible. Effectively, we have replaced each character with a small drawing that looks exactly the same as the character; the plagiarism detection software now sees no text at all, but only diagrams.

The most surprising thing of all is that transforming a PDF in this fashion is easily accomplished with a single command, using only free software:

```
pstoedit -dt -ssp -psarg -r300x300 input.pdf output.pdf
```

The `pstoedit` program⁹ is available on most Linux systems; there is also a Windows version. Our tests, using `pstoedit` v3.45 on Fedora 10, have shown that the above process produces a PDF that is indistinguishable from the original when viewed on screen or printed, but from which no text can be extracted.

The PDF, once thus processed, is not quite ready to be submitted yet. An attempt to submit such a file to Turnitin results in the file's being rejected, not because it contains plagiarized material, but because it is below the minimum acceptable word length: Turnitin cannot see any words at all!

How can we improve on this?

2.3.1 Overlaying text in white

In places where we have applied the Bézier curves technique, and thus made the plagiarized material invisible to the detection software, we can supply replacement text to the detection software, but make it invisible to the human reader. This involves a simple trick: we add some more text, and change its colour to white. Alternatively, in a full typesetting system like L^AT_EX, we could move the replacement text to a position outside the physical boundaries of the page. The detection software will still read the text, in spite of the change in colour or the

⁹<http://www.pstoedit.net/>

out-of-bounds positioning. But the text will not be visible when printing the document, or when viewing it on screen.

By combining Bézier curves with white or mispositioned text, it is possible, if desired, to produce a document whose human-readable text and machine-readable text are completely unrelated.

To supply replacement text, we start by creating another PDF containing the text we want Turnitin to see, and changing its colour to white. We now have two PDFs: one containing the human-readable text, but nothing machine-readable (because it has been converted to Bézier curves), and one containing the machine-readable text, but nothing human-readable (because it is white). The final step is to merge the two.

The merge can be performed in a single step, again using free software. This time we use the `pdftk` program¹⁰, also available on most Linux systems, and also available on Windows. To merge the two files, we execute

```
pdftk bezier.pdf background white.pdf output combined.pdf
```

where `bezier.pdf` is the file containing the Bézier curves, and `white.pdf` contains the white replacement text, and `combined.pdf` is the desired output file.

The resulting file passes through Turnitin without problems. The score given by Turnitin depends entirely on the replacement text, so as long as that is original text, the score will be low. The Turnitin report shown in Figure 6 resulted from submitting a PDF created using `pdftk` v1.41 on Fedora 10. Samples are available online at the URL given in Section 2.1.

3 Analysis

We must ask: how likely is the user to get away with this? Will no-one notice if the user submits a file that has been processed in one of these ways?

3.1 Changing the character map or glyphs

An attempt to extract the text from a document that has had its character map or font altered will result in nonsense text being extracted; this in turn will result in a very low match between the document and any other (probably 0%). This could be picked up on in one of two ways: either the operator suspects foul play because of the 0%, and decides to investigate further, or the operator routinely looks at the extracted text, and notices in this case that the text is nonsense.

Both of these problems can be alleviated by applying the technique to selected parts of the document rather than to the document as a whole. For instance, in a submission with a plagiarized literature review, one could create two fonts, which would look the same, but one would have non-standard character mappings. The literature review could have the technique described above applied to it, using the non-standard font; the rest of the document would be left untouched, using the standard font. The result would be that the plagiarism check would return a much more typical score, and much of the extracted text would look normal. A casual scan through the matches would reveal nothing

¹⁰<http://www.accesspdf.com/pdftk/>

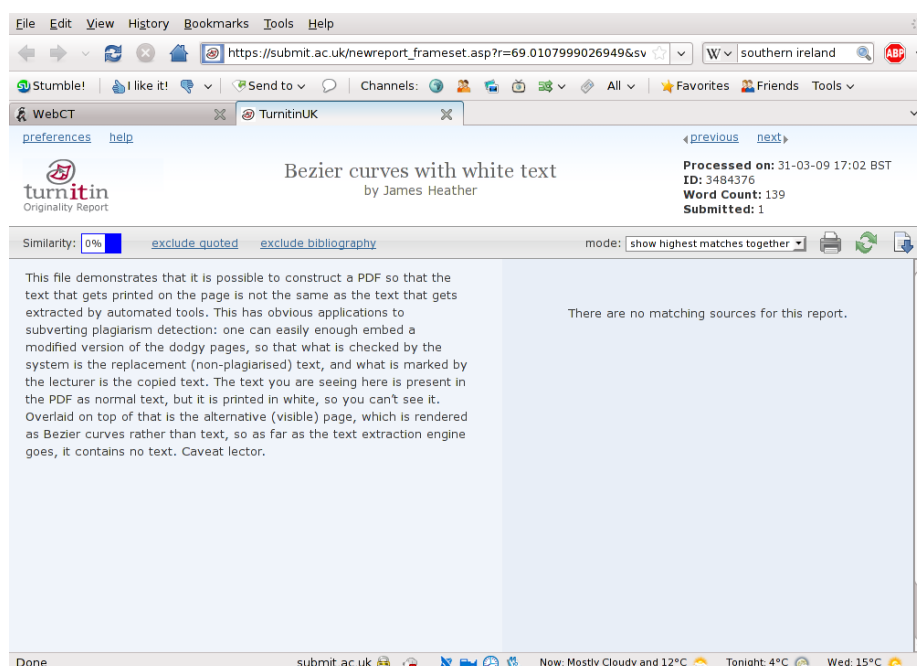


Figure 6: Submitting a PDF with Bézier curves and white replacement text to Turnitin

wrong, because all of the matches found would be from the parts of the text that have not been tampered with.

Our experience is that most academics do not bother to look at the matches if the score returned is unremarkable. Most projects are submitted for a plagiarism check, but then marked on paper if the plagiarism check does not indicate a problem.

3.2 Converting text to Bézier curves

If one converts the document to Bézier curves before submitting, and replacement text is added in white, then the resulting score depends only on the replacement text. This seems very unlikely to be spotted.

Here, too, we can apply the technique to selected parts. Some pages can be converted to curves, and some can be left unchanged; in this case, we do not even need to supply replacement text. If this were applied to a document with a plagiarized literature review, and the plagiarized part were converted to curves, the extracted text would simply have the literature review missing, and the plagiarism detection score would again be somewhere near normal. Although a diligent reading of the extracted text would pick up on the missing text, it seems highly likely that it would go unnoticed. Again, a scan through the matches would not identify anything problematic.

The Bézier curves technique does have one other issue, which is that the process as given above rather inflates the file size. In the original document, the glyph for the letter ‘e’ is stored once only, and referred to many times; after

converting to curves, the glyph is stored many, many times. Whether this would be noticed is open to question: PDFs vary enormously in file size depending on, for instance, whether they contain bitmap images, and if so, whether they have been compressed using a lossy algorithm. A large PDF, particularly in the context of a student project, would usually not raise any eyebrows.

In any case, it would be quite possible to modify the technique to remove this problem entirely. Each character glyph could be converted to curves just once, and stored in the document once; it could then be referred to whenever that glyph is needed. This would produce something analogous to the standard font setup, and would reduce the file size to close to that of the original.

4 Countermeasures

We discuss in this section how Turnitin and other systems can be fixed to defeat these techniques.

It is perhaps worth noting at this point that the problem cannot be reasonably solved by insisting that all submissions are made in Microsoft Word format. This is for three reasons. For one, it imposes severe restrictions on the student, who must now (purchase and) use Microsoft Windows, and (purchase and) use Microsoft Office, even if he or she prefers a different operating system or word processor. For another, Word documents are susceptible to viruses, and can carry hidden information that the author had not intended to reveal¹¹. And finally, it does not defeat the attack: there are many programs and web sites available that offer conversion from PDF to Microsoft Word documents; if Word were mandated for submissions, it would be possible to produce a PDF with Bézier curves as described in Section 2.3, and then use such a facility to convert it into a Word document for submission.

Once all text has been removed from a document, or the text has been altered beyond recognition, there are really only two useful countermeasures. One is to attempt to spot such submissions and alert the operator to the need for a manual check; the other is to use optical character recognition to attempt to recover the text and check it for plagiarism.

4.1 Spotting rogue submissions

The first two techniques (Sections 2.1 and 2.2) produce nonsense text. It would not be difficult for a plagiarism detection system to pick up on this. Small amounts of apparent nonsense would still need to be allowed through, because the nature of the submission might require it: an essay in linguistics, for example, might contain paragraphs in obscure languages, and coursework on cryptography might contain paragraphs of ciphertext. But a nonsense threshold could be set, and any submission exceeding it could be flagged.

For the third technique (Section 2.3), detection is rather more difficult, because there is no nonsense text to spot. The lack of text on particular pages could be flagged, but this is not particularly reliable, because full-page diagrams are not unusual, and it would be difficult to distinguish these from converted text. Certainly if the text is used more selectively yet, to remove specific para-

¹¹<http://news.bbc.co.uk/1/hi/technology/3154479.stm>

graphs from specific pages, it will become extremely hard to detect problems with any degree of accuracy.

4.2 Optical character recognition

In view of these ways of modifying a submission to manipulate the text extraction, and the difficulty of detecting these modifications, it seems that the only reliable way to make certain that the extracted text matches what is represented on the printed page is to use optical character recognition (OCR).

OCR has historically had a reputation for sometimes producing inaccurate results. This, however, is usually down to the quality of the input to the OCR engine. Most of the time, OCR is performed on paper documents that have been scanned; this inevitably comes unstuck when the paper is creased or dirty or when the scanner itself is in need of cleaning. Here, there are no such problems: submissions in the form of a PDF (or similar) can be rendered to a perfect image. A high quality OCR engine should have no trouble interpreting such a document, unless the font used is so unreadable as to make it difficult even for a human to understand. Even if the process misreads the occasional letter, the plagiarism score will not drop noticeably.

The point is that OCR attempts to do the same thing as the human reader of the submission: take a rendered copy of the work and interpret the marks that appear on the page. This immediately counters all attempts to alter the internals of the document. If the document renders acceptably, the OCR engine will be able to read it; if the document fails to render properly, the human marker will not be able to read it, and the problem will be noticed.

4.2.1 Employing OCR to fix the problem

Defeating these attacks using OCR is a fairly simple procedure, and, as with everything else we have discussed, it can be achieved using only free software. It involves additional steps in the text extraction phase along the following lines:

1. Start by recovering text from the PDF in the usual way.
2. Then, use Aladdin Ghostscript to convert each page of the PDF to a bitmap image (at a resolution of around 300dpi).
3. Pass this bitmap through an OCR engine such as Cuneiform or OCRad to recover the text.
4. Identify any text recovered by the OCR engine that was not also present in the text recovered in Step 1.
5. Pass the text from Steps 1 and 4 to the search engine.

In this way, any carefully hidden text is also checked for plagiarism. Since the process in Step 2 is exactly what is done to view or print the PDF, no text can any longer go undiscovered. It no longer matters how the PDF has been manipulated: either the text is visible to the human reader and to the OCR engine, or it is invisible to both.

Although this solves the problems that we have uncovered in current versions of plagiarism detection software, it does come at a small price. Extracting the

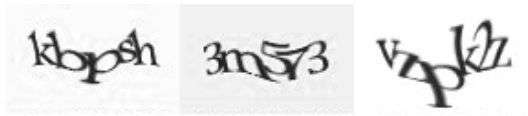


Figure 7: Example of a CAPTCHA

text from a document in the way that is currently employed is fairly trivial and can be done quickly and with minimal impact on the server. Passing the rendered document through an OCR engine, however, is a little more heavy-duty, and would place an increased load on the server. Our tests suggest that it takes around 0.8 seconds of CPU time to extract the text from a single page of a PDF using Cuneiform, a free OCR engine, running on a 3.16MHz Intel Core 2 Duo. If a typical document is around twelve to thirteen pages in length, then this will consume about ten seconds of CPU time for each submission. Turnitin’s own claim on its website is that it expects soon to reach 200,000 submissions per day; at this rate, it would take about 24 cores running flat out to keep up with the workload. Such a solution is far from infeasible, however, for a system of the size of Turnitin.

5 Conclusion

In this paper, we have shown how to defeat current plagiarism detection systems with no financial cost, very little effort, and very little chance of being detected. We have also shown, however, how OCR can be employed to fix these systems to defeat such attacks.

When plagiarism detection systems work, they can be useful and effective at identifying and deterring misconduct. However, when there are loopholes that can be exploited, they can give the operator a false assurance that a submission is original. If submissions are treated as free of plagiarism whenever they are given a low score by the system, either as official policy or simply by assumption on the part of the operator, then the sort of covert plagiarism discussed here will be even less likely to be detected than if there were no system at all.

We believe that the loopholes identified in this paper need to be plugged as a matter of urgency. Fortunately, this is fairly easily accomplished by adopting the OCR approach we have described.

6 Future work

Is it possible to use something like a CAPTCHA (Ahn et al. 2003; Ahn, Blum, and Langford 2004) to produce a PDF that can be read by a human but not by an OCR engine? This is the idea behind the system used on many web sites and in other applications to ensure that the agent using the service is a human and not a spam-bot. Usually the procedure works by warping the text in a way that is easy for a human to decode but difficult for a computer (see Figure 7).

The problem here is rather harder. In its usual form, the CAPTCHA is obviously and visibly distorted. That will not do in this context, because the

distortion will be picked up by the human reader, who will be alerted to the attempt at subverting detection. We would need something strong enough to break the OCR, but subtle enough to look acceptable to the eye. It may be possible to do this by using a font that simulates joined-up handwriting, and keeps the letters very closely packed; this makes it harder to segment the image into individual letters. This is one of the tricks that has been employed in the CAPTCHA in Figure 7—the letters are bunched very closely to make the image segmentation more difficult. Investigating this will be the subject of future research.

Acknowledgements

I am indebted to the anonymous referees, whose comments on an earlier draft of this paper led to substantial improvements in several areas.

Thanks are due to Wolfgang Glunz, creator and maintainer of pstoeedit, for help with tweaking the command-line parameters, and for a useful discussion on how to modify pstoeedit to reduce the file size of the output. I am also grateful to Vladimir Volovich, author of the L^AT_EX `cmap` package, for his helpful explanation of various points.

References

- Ahn, L. v., M. Blum, N. J. Hopper, and J. Langford. 2003. CAPTCHA: Telling humans and computers apart. In E. Biham (Ed.), *Advances in Cryptology, Eurocrypt '03*, Volume 2656 of *Lecture Notes in Computer Science*, pp. 294–311. Springer-Verlag.
- Ahn, L. v., M. Blum, and J. Langford. 2004. Telling Humans and Computers Apart Automatically. *Communications of the ACM* 47(2), 57–60.
- Currie, P. 1998. Staying out of trouble: Apparent plagiarism and academic survival. *Journal of Second Language Writing* 7(1), 1–18.
- Gu, Q. and J. Brooks. 2008, September. Beyond the accusation of plagiarism. *System* 36(3), 337–352.
- Hayes, N. and L. D. Intronas. 2005. Cultural values, plagiarism, and fairness: When plagiarism gets in the way of learning. *Ethics and Behaviour* 15(3), 213–231.
- Pennycook, A. 1996. Borrowing others' words: Text, ownership, memory, and plagiarism. *TESOL Quarterly* 30(2), 201–230.
- Scaife, B. 2007, September. Plagiarism Detection Software Report for Joint Information Systems Committee Plagiarism Advisory Service. Technical report, NCC Group, Manchester. Available online at http://www.jiscpas.ac.uk/documents/resources/PDReview-Reportv1_5.pdf.