Comparing Cookiecutters

ANTONIO SOUSA-DIAS, Queen's University, CISC 499

This paper outlines two metrics for comparison for cookiecutter templates and the development of a search tool, used to provide users with similar templates and explain the functional differences between them. The first metric was based on the structure of the project directory, and the second was based on term frequency of the read me files. These metrics were used to compare the top 300 cookiecutters on GitHub, sorted by stars, pairwise, which was used to assess the precision of each metric. The structural differences between templates along with each templates read me file was used to prompt an LLM to provide a brief summary of the functional differences between each, offering an explanation of which use cases each is better suited for.

1 Introduction

Industrial software projects typically rely on a significant number of packages, libraries, tools, and services that need to be set up and properly configured before development can even begin.

Cookiecutter is a tool used for project templating in order to simplify this process for beginners. It is a mature, open source ecosystem that targets many different domains. Below is a table of some of the most popular cookiecutters:

Name	Domain	Stars	Forks	Files	Dirs
CC Django	Web Apps	12.6k	3k	200	71
CC Data science	Data Science	8.7k	2.5k	38	17
CC PyPackage	Source Code Dev	4.3k	1.8k	31	6
CC CMS	Computational Science	417	91	36	12
govcookiecutter	Data Science	140	36	50	13
CC C++	Source Code Dev	56	6	38	13
CC open edX DevOps	Content Management	44	17	439	119

2 Goal

The intention was to create a way to compare and contrast cookiecutters and offer a measure of similarity between two templates. This could be used to create a search tool for users to find similar cookiecutters, and then prompt an LLM to determine the functional differences between the templates, allowing users to evaluate which is best for their use case.

3 Motivation

The motivation for the project was the hypothesis that a many templates are extremely similar and differ in functionality in subtle ways. Discerning these differences may be difficult for those with limited experiences.

Since cookiecutters are an open source tool, templates are often forked or modified slightly and reuploaded as a different template, which motivated the design of the first metric for comparison.

Term frequency is often used in natural language processing as a measure of similarity. This technique was used for the second comparison metric.

Author's Contact Information: Antonio Sousa-Dias, Queen's University, CISC 499.

4 Metrics

4.1 Structural Metric

The first metric proposed is based on the issue of repositories being downloaded, modified slightly, and reuploaded as a new template. The metric considers the structure of the directory that is built when you run the cookiecutter. The two cookiecutters being compared are built as trees, and the largest common subtree betwen the two is determined, beginning at the root. The number of items in this subtree is counted and divided by the average number of total items between the two structures. This is the conceptual understanding of the metric; in reality, it is computed using set operations to improve efficiency. Below is a formal definition of the metric, where S_1 is the set of files and directories in tree one, S_2 is the set of files and directories in tree two, and s is the similarity of tree one and tree two between $0-1$, with identical trees returning $s = 1$ and completely unrelated trees returning $s = 0$:

$$s = \frac{|S_1 \cap S_2|}{(\frac{|S_1|+|S_2|}{2})}$$

4.1.1 Representing Structures

The directory structures are built as trees recursively using nested dictionaries. Starting at the slug file present in all cookiecutters, usually named {{cookiecutter.project_slug}}, or something similar.

The files are added to a dictionary named 'files' where each file name points to another dictionary of the properties of that file. In the final version, the only property that was stored was the sha hash of the file contents, but it was done this way to allow for the ability to scale up if more properties were needed.

Below is an example of this structure representing a single file:

```
"files": {
    "file": {
        "sha": "hash"
        }
    }
```

The subdirectories are added to a dictionary named 'dirs' and the names of the subdirectories point to the result of recursively building the structure from that subdirectory.

Below is an example of a small tree representing a directory structure and the corresponding nested dictionary:

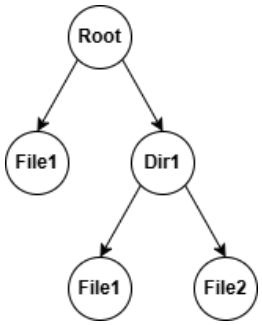

Fig. 1. Tree representing a simple directory structure

```
{
  "dirs": {
    "dir1": {
      "dirs": {
      }
    },
    "files": {
      "file1": {
        "sha": "hash1"
      },
      "file2": {
        "sha": "hash2"
      }
    }
  },
  "files": {
    "file1": {
      "sha": "hash1"
    }
  }
}
```

4.1.2 Largest Common Sub-tree

In order to get the similarity score based on the structure of the templates using set operations, a function was made to create sets of the unique directories and files in a given structure. The function is recursive in nature, and traverses the nested dictionaries, keeping track of the current path by accepting a string representation of this path as a parameter. This allowed for duplicate files and directories in different locations to be considered as separate entities, and enabled the structures to be compared without fully building the largest common sub-tree. The individual files are stored as tuple of length 3, containing the path, name, and hash, in that order. Directories are stored as a tuple of length 2, containing the path and the name, in that order. Below is the tuple of File2 and Dir1 in the previous example:

```
("root\dir1", "file2", "hash2") & ("root", "dir1")
```

These sets are created and the set intersection is taken, representing the largest common sub-tree, which can be built and represented, if desired, as a nested dictionary by recursively iterating through this set, beginning at the root. The size of these sets can be found using the built in len() function and used in the similarity formula defined in the previous section.

4.1.3 Diffing

Finding this common sub-tree naturally leads to finding the collection of unique sub-trees in each structure. Conceptually, this is obtained be 'subtracting' the largest common sub-tree from each larger tree. This can be useful to manually determine differences between large templates, or, as

will be discussed later in this paper, to prompt an LLM to summarize functional differences between the templates. The process of finding these uncommon forests, which will be referred to as diffs going forward, will be outlined.

Similar to the process of finding the largest common sub-tree, the conceptual understanding is a convenient way to frame the problem, but the process of actually finding the diffs is achieved using the sets of tuples and set operations.

The first step in this process is to take the difference of the sets of directories and files and store them as new sets. This is formalized below, where F_1 is the set of files in tree one, D_1 is the set of directories in tree one, F_2 is the set of files in tree two, and D_2 is the set of directories in tree two:

$$F_{1,2} = F_1 - F_2$$
$$D_{1,2} = D_1 - D_2$$

$D_{1,2}$ is iterated through to find the set of directories whose paths are in the common sub-tree. This pinpoints the locations where the structures start to differ, without double counting compounding differences. These paths are added to a set, R_D, and are iterated through to build each sub-tree in the diff recursively, starting from each element in R_D, only including items in $D_{1,2}$ and $F_{1,2}$.

The file paths are found separately, after this initial diff is built. This is because if a file is in a directory that is unique to a particular structure, then it will inherently be included when the structure is built from a path that is above the file in the tree.

There is still the edge case of a file being changed or added to a directory that does exist in both structures, and whose subdirectories are unchanged. The paths of these instances are found by iterating through $F_{1,2}$, while checking if any of the element's sub-paths are contained in the set R_D. This is to ensure that the file is not below a directory change that has already been accounted for. If no matching sub-path is found, and the path has not already been included in R_F, then the path is added to the set R_F.

R_F is then iterated through to build the sub-trees where only files have been altered, and are therefore only of depth one, including only files in $F_{1,2}$. These are then added to the already constructed diff as disconnected components of the larger forest.

It is important to note that this process is not symmetric since set subtraction is not commutative. This aligns well with the conceptual understanding of the problem as subtracting the largest common sub-tree from one of the original trees - one would not expect the results to be the same regardless of which of the original trees is chosen to be subtracted from. This means that finding the diffs between two templates requires performing these operations twice.

4.1.4 Example

Consider comparing the tree from the earlier example with a second tree shown here:

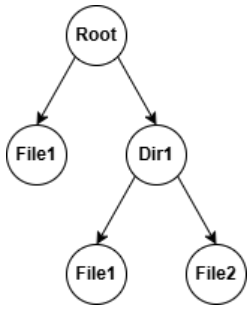

Fig. 2. Tree One

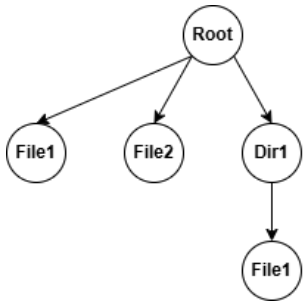

Fig. 3. Tree Two

The following are the sets of files and directories of each respective tree:

$$S_1 = \left\{ \begin{array}{l} \texttt{("root", "file1", "hash1")}, \\ \texttt{("root/dir1", "file1", "hash1")}, \\ \texttt{("root/dir1", "file2", "hash2")}, \\ \texttt{("root", "dir1")} \end{array} \right\} \quad S_2 = \left\{ \begin{array}{l} \texttt{("root", "file1", "hash1")}, \\ \texttt{("root", "file2", "hash2")}, \\ \texttt{("root/dir1", "file1", "hash1")}, \\ \texttt{("root", "dir1")} \end{array} \right\}$$

The common elements can be found by taking the intersection of these sets:

$$S_1 \cap S_2 = \left\{ \begin{array}{l} \texttt{("root", "file1", "hash1")}, \\ \texttt{("root/dir1", "file1", "hash1")}, \\ \texttt{("root", "dir1")} \end{array} \right\}$$

Which corresponds to the following largest common sub-tree:

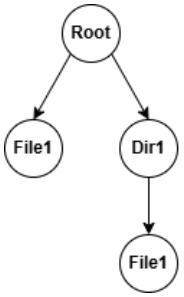

Fig. 4. Largest Common Subtree

The sets of unique elements can be found with set subtraction. It should be noted that $S_1 \cap S_2$ does not be computed for this step, although in this context it usually will be:

$$S_1 \setminus S_2 = \{ \texttt{("root/dir1", "file2", "hash2")} \}$$

$$S_2 \setminus S_1 = \{ \texttt{("root", "file2", "hash2")} \}$$

These sets correspond to the following trees:

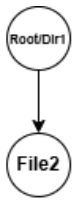

Fig. 5. Diff One

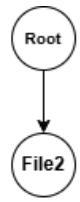

Fig. 6. Diff Two

These trees would be returned in the following format:

```
{
  "root/dir1": {
    "dirs": {
     },
    "files": {
      "file2": {
        "sha": "hash2"
      }
    }
  }
}
```

```
{
  "root": {
    "dirs": {
     },
    "files": {
      "file2": {
        "sha": "hash2"
      }
    }
  }
}
```

4.2 TF-IDF Metric

Natural language processing, or NLP, is an area of computer science that is concerned with enabling computers to understand human language and the meaning behind it. Common strategies in this field, specifically 'term frequency-inverse document frequency', or TF-IDF, were used to create a second metric which contrasts the much more precise structural metric previously outlined. The goal was to create a metric that captured similarity using techniques that approach the problem from the other end of the spectrum in terms of rigor.

4.2.1 TF-IDF

TF-IDF is a method used to evaluate the importance of a word in a given piece of text. It weighs the occurrences of a word in a given document with the number of times it appears in the other documents whose contents is included in the vocabulary. This is an intuitive notion of importance.

If a word occurs frequently in a particular document, but also frequently in every other document, it does not provide much information as to what the document is about; however, if only a few documents contain this word, its presence in a given document contains much more information about the contents of that document.

4.2.2 Vectorization

Using the SciKit library, the read me files of a collection of templates can be vectorized. This means that for each file, values for each word in the vocabulary are produced using the TF-IDF technique. This results in a vector residing in a vector space with dimensionality equal to the size of the vocabulary. It is intuitive that this would result in read me files with similar content 'pointing in similar directions'.

4.2.3 Cosine Similarity

Again, using the SciKit library, the cosign similarity between each vector can be calculated. This is done by computing the cosign of the angle between two vectors, which results in a value between -1 and 1, offering a measure of how similar two vectors are. If the angle between the vectors is 0, meaning the vectors lie on top of each other, the cosine similarity is 1. If the vectors are perpendicular to each other, the cosine similarity is 0, and if they point in opposite directions, the cosine similarity is -1. This is a difficult concept to visualize in high-dimensional vector spaces, but is simple to grasp in two dimensions and then abstracted into these higher dimensions.

Fig. 7. Similar Vectors

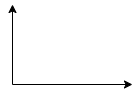

Fig. 8. Unrelated Vectors

Fig. 9. Opposite Vectors

4.2.4 Processing Read Me Files

The read me files of the templates were often formatted as markdown files, therefore, some preprocessing was required in order to improve the performance of the vectorization.

The code blocks were removed and the plain text was retrieved. The text was converted to all lower case and the numbers and punctuation were removed. Finally, as a parameter to the SciKit vectorizer, stop words were removed. These are words that are too common to encode any meaning, such as 'the', 'a', 'is', ect.

4.3 Advantageous & Disadvantageous

The structural metric is a very literal method of comparison that relies on recycled and iterated templates occurring relatively frequently to be useful. Additionally, even if this does occur often, the metric can still only detect similarity in 'related' templates. Two templates can target the same domain in similar ways and be determined to be completely dissimilar according to this metric if they do not exactly share any components.

The TD-IDF metric addresses this issue, but also relies on templates to be well documented in order to be effective. Many read me files are very scant and do not offer much detail about the project, resulting in poor performance. This is where considering the contents of the project itself is more useful.

5 Implementation

In order to evaluate these metrics, a dataset needs to be constructed. This was done by scraping the contents of top 300 most popular cookiecutters on github, sorted by stars. This number was chosen

to get a sufficiently large sample size, while maintaining the feasibility of later comparing the templates pairwise with eachother. The templates were sorted by popularity to get a representative subset of the available cookiecutters.

5.1 Structural Metric

The contents of each repository was iterated through, until the previously mentioned slug file was found, and the nested dictionary was then built with this directory as the starting point. Each structure was saved locally as a json file to be retrieved later.

The 300 structures were compared pairwise with each other using the structural metric. This means that there was $\binom{n}{k}$ comparisons made, which simplifies to 44850, illustrating the importance of opting to use set operations to avoid directly constructing the largest common sub-tree. Additionally, the diffs are not computed at this stage.

5.2 TF-IDF

The contents of the read me files for each repository was also scrapped, cleaned, and saved locally as a json file. These were then processed at once, using the SciKit library, resulting in a 300×300 symmetric matrix where each entry i, j represented the similarity score of the i-th and j-th entry when compared using cosine similarity.

6 Evaluation of Metrics

These results were used to evaluate the precision of each metric. The top 150 most similar pairs of templates according to each metric were returned for manual inspection.

6.1 Structural Metric

Below is an excerpt from the output of the structural metric that was used to assess its precision:

```
('chris1610_pbp_cookiecutter', 'mkrapp_cookiecutter-reproducible-science')
('associatedpress_cookiecutter-python-project', 'associatedpress_cookiecutter-r-project')
('Jean-Zombie_cookiecutter-django-wagtail', 'julianwagle_earnalotbot')
('audreyfeldroy_cookiecutter-pypackage', 'elgertam_cookiecutter-pipenv')
```

These pairs were labeled as true positives and false positives based on manual inspection, considering the github pages, documentation, and fork history. These four pairs were sorted as follows:

```
True Postives:
('chris1610_pbp_cookiecutter', 'mkrapp_cookiecutter-reproducible-science')
('audreyfeldroy_cookiecutter-pypackage', 'elgertam_cookiecutter-pipenv')

False Positives:
('associatedpress_cookiecutter-python-project', 'associatedpress_cookiecutter-r-project')
('Jean-Zombie_cookiecutter-django-wagtail', 'julianwagle_earnalotbot')
```

6.2 TF-IDF

Below is an excerpt from the output of the TF-IDF metric that was used to assess its precision:

```
('wemake-services_wemake-django-template', 'wemake-services_wemake-python-package')
('senseta-os_senseta-base-project', 'tiangolo_full-stack')
('microsoft_cookiecutter-spacy-fastapi', 'microsoft_python-sklearn-classifier-cookiecutter')
('MihailCosmin_cookiecutter-python-gui-application', 'mandeep_cookiecutter-pyqt5')
```

These pairs were labeled as true positives and false positives in the same way as the previous section. These particular pairs were classified as follows:

```
True Postives:
('senseta-os_senseta-base-project', 'tiangolo_full-stack')
('MihailCosmin_cookiecutter-python-gui-application', 'mandeep_cookiecutter-pyqt5')

False Positives:
('wemake-services_wemake-django-template', 'wemake-services_wemake-python-package')
('microsoft_cookiecutter-spacy-fastapi', 'microsoft_python-sklearn-classifier-cookiecutter')
```

6.3 Analysis

These results are illustrative of a theme in the errors made by each metric. The structural metric was more conservative and therefore correct more often. This follows intuitively from the understanding of the metric. It is hard to argue that there is no level of similarity when files and directories are common between two templates. Additionally, it is unlikely that a user would iterate on a template to target a different domain than the original. The false positives returned by this metric were often due to one of two reasons - templates that were made by the same author or templates that were overly specific which prevented it from being generally useful. These two cases were seen in the earlier example. It is understandable that an author may use the same starting point for their templates, even if those that are designed for different domains, but it is still incorrect to determine that these templates can be classified as being similar. The issue of overspecification is one that is clearer when viewed from the perspective of a user searching for similar templates - it is not useful for these templates to be returned to a user if they are not interested in that specific use case of a more general domain.

The TF-IDF metric was able to identify similarity in completely unrelated templates, in terms of contents, as expected. This meant that the metric was less precise but offered more variety and specificity in terms of the domains targeted by the pairs returned. The structural metric returned mostly templates in very common and general domains, and the pairs almost formed families of templates that were all iterated on from a single template. This was seen in the examples, and follows from the fact that more specific use cases have less templates and therefore it is less likely these templates would be related to each other. Common authors was also an issue for the TF-IDF metric, since the names of the authors were included in the dictionary if they were mentioned in the read me, so even if the name only appears once in a read me, the low document frequency would cause this inclusion to be very significant.

6.4 Results

Totaling the number of true positives and dividing by the number of total pairs returned a precision rate for each metric. The structural metric returned 139 total true positives out of 150 total pairs returned, giving a precision rate of 92.67%. The TF-IDF metric returned 118 total true positives, giving a precision rate of 76.67%. Considering the similarity score of the 150th pair returned by each metric gives a threshold beyond which these levels of precision can be assumed. For the strucural

metric, this score is 0.3, and for the TF-IDF metric, this score is 0.175. These results are summarized in the following table:

Metric	True Positives	Precision (%)	Threshold Score
Structural	139	92.67%	0.3
TF-IDF	118	76.67%	0.175

Table 1. Precision rates and similarity thresholds for each metric, based on the top 150 returned pairs.

6.5 Clustering

Both metrics were used to create a distance matrix as the input to a hierarchical clustering algorithm, HDBACAN. This was once again implemented using the SciKit-Learn library. Upon manual inspection of the clusters generated, the TF-IDF metric is preferable for this use case, as the clusters formed by the structural metric were inconsistent and often included unrelated templates. This is a result of the discontinuous nature of the metric; there are many pairs of templates that do not share any files or directories and are therefore deemed completely dissimilar, resulting in a sparse distance matrix. TF-IDF metric is more nuanced and can identify very subtle potential similarities between templates, which results in more cohesive clusters.

7 Search Tool Example

A search tool was implemented using the metrics. The workflow of using the tool will be outlined.

The user provides a cookiecutter, in this case it is a data science project that is being considered:

```
'mkrapp/cookiecutter-reproducible-science'
```

The search tool returns the three most similar templates according to the structural metric, passed a certain threshold, and then the TF-IDF metric, if there are less than three initially returned:

```
'jeannefukumaru_cookiecutter-ml': 0.89
'daveebbelaar_df-data-science-template': 0.58
'chris1610_pbp_cookiecutter': 0.39
```

The user can selected which of these templates to further compare, and it will be diffed with the users provided template. The resulting diffs, the largest common subtree, and the read me files are provided to an LLM that is prompted to describe the functional differences between each template. Below is the prompt that was used:

```
def interpret(lcc, diff1, diff2, readme1, readme2):
    prompt = (
        'Given the following largest common subtree:\n'
        f'{json.dumps(lcc, indent=4)}\n\n'
        'And the following differences in repo one:\n'
        f'{json.dumps(diff1, indent=4)}\n\n'
        'And the following read me for repo one:\n'
        f'{readme1}\n\n'
        'And the following differences in repo two:\n'
        f'{json.dumps(diff2, indent=4)}\n\n'
        'And the following read me for repo two:\n'
        f'{readme2}\n\n'
        'Please interpret what differences in functionality exist between these
        repositories which are both cookiecutters aimed at project templating.'
    )
```

Below is example output from comparing the users template with `'chris1610_pbp_cookiecutter'`:

```
'mkrapp/cookiecutter-reproducible-science' is focused on
scientific workflow, encouraging reproducibility
and a structured project lifecycle suitable for
comprehensive data science projects in research settings.

'chris1610_pbp_cookiecutter' serves users looking for
a straightforward approach to setting up data
analysis projects, with a clear focus on using Jupyter
notebooks and offering simplicity in project structuring.
```

The idea was that providing the structures would allow the LLM to interpret the file names, extensions, and directory locations, and then combine this knowledge with the contents of the read me files, in order to determine the use cases that each template is better suited for.

In this example the LLM was successful in detecting such subtleties, below is another excerpt of output illustrating this:

```
'mkrapp/cookiecutter-reproducible-science': Contains directories
such as `reports`, `data` (with subdirectories for
`interim`, `raw`, `processed`, and `external`),
`notebooks`, `config`, `bin`, `docs`, and `src`
(which includes subdirectories for `tools`, `external`,
`models`, and `visualization`). This suggests a comprehensive
structure designed to handle various stages of a data science
project from raw data acquisition to analysis, modeling, and reporting.

'chris1610_pbp_cookiecutter': Includes a more simplified directory
structure focusing on `root`, which holds `.ipynb_checkpoints`
and notebook files. This indicates a primary focus on the
analysis and prototyping stages of a data science project rather
than a more comprehensive project lifecycle.
```

This is an accurate assessment of the functionality of each of these templates.

8 LLM Interpretation

More generally, the LLM interpretations were occasionally useful, but often suffered from forms of hallucinations. The most promising use case for the LLM interpretations is to summarize templates with verbose read me files, as LLMs excel at this task. The problems occurred when the LLM was prompted with a template that was poorly documented; it was unable to consistently interpret the structures provided and therefore struggled to expand on such a template.

9 Future Work

Due to time constraints, there were many ideas that could not be executed. The most intuitive next step given more time would be to expand the number of templates considered and the number of pairs used to evaluate the precision of each metric. This would allow for more confident conclusions to be made about the quality of both metrics.

The structural metric could be improved by manually labeling pairs of similar templates and using a neural network to learn structural features that contribute most significantly to a templates functionality. This technique could be combined with more nuanced structures to further improve the metric.

An idea that was considered was to use locality sensitive hashing instead of the sha hash that was implemented. This would have the effect that similar files would have similar hashes, instead of having hashes that did not contain any information about the contents of the file. These hashes could be used to match to the most similar files in the templates being compared.

The contents of the files could be further considered by using word embeddings and storing the vectors of each files as attributes. The cosine similarity could then be used to assess how similar the contents of certain files are. This would remedy the problem of only directly related templates being considered similar according to the structural metric.

The LLM interpretation could be improved by prompt engineering and providing more context for the LLM to parse, instead of relying solely on the structure, file names, and file extensions. This could be achieved by providing important excerpts of the source code or providing more attributes about each template, and could drastically improve the LLM's performance in cases where the templates are poorly documented.

10 Conclusion

This paper proposed two metrics for comparing cookiecutter templates, which could potentially be successfully extended to GitHub projects in general. The first metric was based on the idea that these templates are iterated on, and re-uploaded with slight modifications to their functionalities that are potentially difficult for a user to detect. Therefore, this metric considered the structure of the project, finding the number of items in the largest common sub-tree between two templates being compared. The second metric used the contents of the read me files and TF-IDF for more flexible comparison of templates. Both metrics were manually evaluated and found to have a precision of 92.67% and 76.67% respectively.

TF-IDF is a well established technique, so the more interesting conclusion is the impressive effectiveness of the structural metric at identifying similar features in templates, corresponding to their use cases. It can also be concluded that the iterating and reuse of templates occurs relatively frequently with cookiecutters, otherwise this metric would be completely ineffective. Additionally, there were very few templates that were overly similar to other templates, at least in the top 300 most popular, indicating that users are successfully filtering for useful templates, suggesting that the open source cookiecutter ecosystem is healthy.

There are many iterations that could be made to each metric in order to improve their respective performance and flexibility; it is potentially a very useful endeavor that could extend beyond the context of cookiecutters, and the success of this limited implementation is very promising.

References

[1] J. Dingel, A. Sun, H. Elabd, N. Yao, A. Boulos, and A. Tizghadam. Project templating and onboarding with cookiecutter: Foundations, uses, and guidelines. 2023. Version 2023, unpublished.

[2] OpenAI. Openai api. 2024.

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