Multi-Store Metadata-Based Supervised Mobile App Classification

Giacomo Berardi, Andrea Esuli, Tiziano Fagni
Istituto di Scienza e Tecnologie dell'Informazione
Consiglio Nazionale delle Ricerche
56124 Pisa, Italy
E-mail: {firstname.lastname}@isti.cnr.it

Fabrizio Sebastiani*
Qatar Computing Research Institute
Qatar Foundation
Doha, Qatar
E-mail: fsebastiani@qf.org.qa

ABSTRACT

The mass adoption of smartphone and tablet devices has boosted the growth of the mobile applications market. Confronted with a huge number of choices, users may encounter difficulties in locating the applications that meet their needs. Sorting applications into a user-defined classification scheme would help the app discovery process. Systems for automatically classifying apps into such a classification scheme are thus sorely needed. Methods for automated app classification have been proposed that rely on tracking how the app is actually used on users' mobile devices; however, this approach can lead to privacy issues. We present a system for classifying mobile apps into user-defined classification schemes which instead leverages information publicly available from the online stores where the apps are marketed. We present experimental results obtained on a dataset of 5,993 apps manually classified under a classification scheme consisting of 50 classes. Our results indicate that automated app classification can be performed with good accuracy, at the same time preserving users' privacy.

1. INTRODUCTION

Mobile devices such as smartphones or tablets are widely used nowadays. In this sector the software distribution model is radically different from the standard distribution model of computer software. Mobile applications are mainly available, if not exclusively, on specific channels called "markets" or "stores". The user experience is then simplified: users can browse these stores, locate specific apps, and download them immediately, which means that the software is directly transferred from the producer to the consumer. Each mobile platform, such as Android or iOS, has its own store (e.g., the Play Store for Android apps and the Apple Store for iOS apps), which is usually accessible from the Web or from a dedicated app. However, many new stores are springing up for each platform, especially for Android[1]. This fragmentation of the software distribution channels may negatively affect the user experience.

In order to help the user in locating the apps she needs, stores are internally organized according to a pre-determined classification scheme. That is, each app is labelled according to one or more classes it belongs to, which helps the user in exploring the store more effectively. However, it is the developers themselves which classify their own apps; especially in stores with permissive publication policies, this may mislead users. Additionally, each store relies on its own classification scheme, depending, for example, on the functionality of the store and the target for which it is intended; for instance, the Amazon Appstore only hosts apps for the Kindle eBook reader, which means its classification scheme is targeted more to readers than to generic users. In a scenario in which the user wants to explore and find an app suitable for her own needs, it would instead be more useful to have a unique, user-defined classification scheme according to which all apps, independently of the store they are marketed on, are classified.

In this paper we present a system for automatic app classification. Methods for automated app classification have been proposed that rely on tracking how the app is actually used on users' mobile devices [5]; however, this approach can lead to privacy issues. Our solution enables users to automatically classify apps under a user-defined classification scheme, independently of the app's distribution platform, at the same time preserving users' privacy. We tackle automated app classification by leveraging the metadata (textual and numeric data) contained in the page that describes the app (which is thus identified by its URL), as contained in a specific app store. We approach the app classification task through supervised learning: given a custom pre-defined classification scheme and a set of apps manually classified according to it, our system trains (using this set of apps as training data) a classifier capable of automatically assigning classes in the classification scheme to new, unseen apps. We assume that each app can be assigned to one or more classes, which make this an instance of *multi-class multi-label classification* ("multi-class" referring to the fact that there are more than 2 classes in the classification scheme, "multi-label" referring to the fact that more than one class

*Fabrizio Sebastiani is on leave from Consiglio Nazionale delle Ricerche.

[1] http://en.wikipedia.org/wiki/List_of_mobile_software_distribution_platforms

can be attributed to the same app).

Classifying apps can be seen as a subtask of *app discovery*[2]. Stores contain several hundred thousands apps, and the process of searching for the desired app (i.e., the one with the right functionality) is more complex than retrieving simple textual documents. Supporting discovery with classification is already employed by popular stores such as the Apple Store and Google Play, which have recently expanded their classification scheme to a richer set of classes and subclasses[3]. Mobile apps and services for app discovery are becoming popular. One example is Xyo[4], which creates a stream of recommended apps; a user can navigate through the stream, and each app description is enriched with subclasses automatically assigned by Xyo. Appcrawlr[5] is another service which makes use of machine learning and NLP technologies in order to capture app genres and functionality. All these developments in app discovery date back to last year; this task is going to be central in the growth of the stores, and the tools for classifying apps are consequently becoming of key importance.

2. A SYSTEM FOR APP CLASSIFICATION

2.1 Crawling App Stores

The first step of the process consists in gathering the metadata of the mobile applications we are interested in. In this work we focus on Android and iOS apps, disregarding for the moment apps for the Windows Phone. The metadata of a specific application can be retrieved from the Google Play store[6] or the Apple iTunes store[7], depending on which stores the app is published on. To access metadata of an Android application, we use the open source Java library 'marketapi'[8]. Given an Android application unique address (e.g., the Gmail app address is `com.google.android.gm`), the library, by querying Google servers, allows to quickly retrieve the various metadata fields about the application that are relevant for our purposes (see Section 2.2). In order to retrieve metadata about iOS applications, we use the public Apple service 'Search API'[9]. This Web REST API can be queried by submitting the unique application ID (e.g., `https://itunes.apple.com/lookup?id=5654554`), which results in a JSON response containing all metadata related to that application. The JSON response can then be easily analysed to extract all information of interest.

2.2 Content Extraction

App metadata are processed in order to extract features to be fed to the learning algorithm. The metadata fields from which we extract features are the following:

- **Description**: This field briefly presents the app, and it usually lists its characteristics and functionality. This information, expressed in natural language, is relevant for our task not only because of the explicit clues it provides (e.g., "IMAP e-mail client"), but also because of the stylistic and genre-specific use of language that can characterise some classes of apps (e.g., adventure games are often described using an emotion-laden language, while music-streaming apps often cite famous artists' names). We extract all the words contained in the "description" field and perform "stop word removal" (i.e., we remove content-neutral words such as articles and prepositions) and "stemming" (i.e., we map a word into its morphological root, so as to collapse e.g., "computers" and "computational" into the same feature).

- **AppleClassScheme**: We include the Apple Store class assigned by the app developer as a feature. Each app is assigned exclusively to one class (e.g., Business, Education, Games, ...) in the store, so this yields a single feature whose value is the app's class name.

- **GoogleClassScheme**: Similarly as for "AppleClassScheme", we include the Google Play store class assigned by the app developer as a feature.

- **Name**: Given the name of an app, we extract all the words contained in it and perform stop word removal and stemming.

- **AverageUserRating**: User ratings define the quality of an application, and they can be used as a clue, e.g., to detect scam applications; for instance, for an app with a low AverageUserRating we might not want to trust the developer-provided information in the AppleClassScheme / GoogleClassScheme field. We extract the average user rating of each application, and we define a numerical feature which takes values in the $[0, 5]$ interval, the greater the value the higher the rating.

  It should be mentioned that we do not use the total number of downloads as an additional feature, since this number is known to be strongly correlated to average user ratings [1], and would thus likely represent duplicate information.

- **UserRatingCount**: The number of user ratings defines the popularity of an application. Together with AverageUserRating this feature type represents user-related information, as it quantifies the feedback an app receives. Again, this is a single numerical feature with positive integer values.

- **FileSize**: The app file size is an indicator of the complexity of the application, so it seems a good feature for discriminating apps with different targets of use (e.g., the file size of games is usually large). Again, this is a single numerical feature with positive integer values (number of bytes).

We qualify all the features we extract by a prefix that indicates which field the feature comes from, with the goal of placing different emphasis on features coming from different fields. For instance, word "maps" may have less importance if it is extracted from the Description field (in which case it is represented as `DESCRIPTION:maps`) and more importance if it is extracted from the Name field (in which case it is represented as `NAME:maps`).

[2]http://goo.gl/QvzBDx
[3]http://bit.ly/1xtFV7F
[4]http://goo.gl/07Q8Jy
[5]http://appcrawlr.com/app/technology
[6]https://play.google.com/store
[7]http://www.apple.com/itunes/overview/
[8]https://code.google.com/p/android-market-api/
[9]http://goo.gl/iYaIcA

We analyse, by means of the Language Detection library[10], each app description in order to determine the language it is written in. For the purposes of this experimentation we discard from consideration all apps whose description is deemed not to be in English.

2.3 Feature Selection and Weighting

The feature extraction process described in Section 2.2 returns a high number of features. For the dataset that we use in the present work (see Section 3 for details), no less than 50,371 features are generated from the 5,792 apps the dataset consists of. In order to keep the computational effort to a more manageable level we perform *feature selection* (FS). This latter consists in identifying a subset $S \subset T$ of the original feature set T (which coincides with the set of features that occur in at least one training instance) such that $|S| \ll |T|$ and such that S reaches a compromises between (a) the efficiency of the learning process and of the classifiers (which is inversely proportional to $|S|$), and (b) the effectiveness of the resulting classifiers. We perform feature selection by (a) scoring each feature t_k according to its estimated contribution to discriminating class c_j from the other classes, and (b) retaining only the highest-scoring ones. As the scoring function we use *information gain*, defined as

$$IG(t_k, c_j) = \sum_{c \in \{c_j, \overline{c}_j\}} \sum_{t \in \{t_k, \overline{t}_k\}} P(t,c) \log_2 \frac{P(t,c)}{P(t)P(c)} \qquad (1)$$

Only the x features with the highest $IG(t_k, c_j)$ value are retained. We then adopt a "round robin" policy [2] in which the n (internal and leaf) classes take turns in picking a feature from the top-most elements of their class-specific orderings, until x features are picked.

All the selected features are weighted by means of the well known BM25 weighting function [4]. As the learning algorithm we use support vector machines (SVMs), and more specifically the implementation provided by the freely available `libsvm` software package[11]. Since `libsvm` requires features to be numeric, we convert set-based features (such as "AppleClassScheme" and "GoogleClassScheme") into binary ones. For instance, feature "AppleClassScheme" has 46 possible values, since the classification scheme used in the Apple Store consists of 46 classes and each app is classified under exactly one of them. We thus feed `libsvm` with 46 binary features, each indicating whether the app has the corresponding class or not. We train a binary classifier on each class of the classification scheme. More than one classifier can return a positive decision for the same app, i.e., this classification task has a multi-label character. In `libsvm` we use a linear kernel with the parameter `C` (the penalty of the error term) set to 1; this configuration, while simple, achieves a good trade-off between model complexity and effectiveness, and is the default setting used in `libsvm` software.

3. EXPERIMENTAL SETTING

For our experiments we have used a classification scheme and a dataset that were provided to us by a customer. Note that we had no control on the design of the classification scheme and on the choice of the dataset; we thus take both as given[12]. The dataset consists of 5,792 app IDs classified according to a flat classification scheme consisting of 50 classes; each app belongs to $1 \leq n \leq 4$ classes. The dataset is very imbalanced, since a single class ("Games") accounts for 4,903 apps (84.6% of the total), while the other 49 classes have just an average of 34.2 apps each. This makes learning accurate classifiers for these 49 classes a difficult problem.

Table 1: Accuracy of classifiers obtained by using all extracted features (Columns 2 to 4) or by using selected features only (Columns 5 to 7). In Column 5, percentages indicate the survival rate of the specific feature type. Boldface indicates best results.

	No Feature Selection			With Feature Selection		
	#Features	F_1^M	F_1^μ	#Features	F_1^M	F_1^μ
Description	45652	0.219	0.878	14076 (30.8%)	0.229	0.880
AppleClassScheme	46	0.118	0.862	29 (63.0%)	0.048	0.816
GoogleClassScheme	38	0.128	0.853	30 (78.9%)	0.095	0.834
Name	4632	0.246	0.849	968 (20.9%)	0.187	0.826
All feature types	50371	0.311	**0.895**	15106 (30.0%)	**0.320**	**0.895**

We perform feature selection and retain 30% of the original features, bringing the original number of 50,371 features to a more manageable number of 15,106 features.

As a measure of effectiveness that combines the contributions of *precision* (π) and *recall* (ρ) we use

$$F_1 = \frac{2\pi\rho}{\pi + \rho} = \frac{2TP}{2TP + FP + FN} \qquad (2)$$

where TP stands for true positives, FP for false positives, and FN for false negatives. We average the class-specific F_1 scores across the classes by computing both microaveraged F_1 (denoted by F_1^μ) and macroaveraged F_1 (F_1^M). F_1^μ is obtained by (i) computing the class-specific values TP_i, (ii) obtaining TP as the sum of the TP_i's (same for FP and FN), and then (iii) applying Equation 2. F_1^M is obtained by first computing the F_1 values specific to the individual classes, and then averaging them across the c_j's.

4. RESULTS

Table 1 presents the results from our experiments; all results were obtained by 10-fold cross-validation (10FCV).

A fact that emerges from the microaveraged results (F_1^μ) is that we have obtained very high accuracy for both the NFS and WFS configurations. The macroaveraged results (F_1^M) are lower, but this was to be expected in the light of the severely imbalanced nature of the dataset. In fact, the micro- and macro-averaged versions of a measure yield the same value only when the dataset is perfectly balanced, while the former yields higher values than the latter when the dataset is imbalanced. In our case, the most frequent class (Games) totals 4,902 instances while the least frequent one (Tethering) has only 1 instance, which is an indication of the severe level of imbalance of this dataset.

A second fact that emerges is that feature selection not only does not harm accuracy, but even gives some marginal accuracy improvements in terms of F_1^M, which indicates that

[10]`http://code.google.com/p/language-detection/`

[11]`http://www.csie.ntu.edu.tw/~cjlin/libsvm/` .

[12]Unfortunately we have no rights on the data and on the classification scheme, so we are not in a position to make them publicly available.

many features that had originally been extracted are useless for the classification process. Using feature selection is thus a win-win approach, also due to the fact that the computational cost of both the learning and the classification phases scale linearly with the number of features used [3].

4.1 Effects of Different Feature Types

In an attempt to better understand the quality of the features contributed from each of the fields identified in Section 2.2, for each such field we have run experiments by using the features extracted from that field only; the results are reported in Table 1, each row representing a specific feature type. While these experiments do not account for the subtle interactions that may take place as a result of the copresence of features of different types, they nonetheless give an indicative idea of the relative contribution of the different types. We have run such experiments (a) with the full feature set (Columns 2-4 of Table 1), and (b) with the set resulting from the feature selection process (Columns 5-7 of Table 1). The feature selection process consisted in putting into a common pool all the extracted features of all types, and then selecting the best ones irrespective of their type, so that different feature types compete with each other[13]. In Column 6 we report, for each feature type, the *survival rate* of a given feature type, i.e., the percentage of features of that type that made it into the final selected set.

As expected, "All feature types" is the best setting overall, i.e., no single feature type outperforms it. While this is true for both measures, this is particularly evident for F_1^M, both in the NFS and WFS configurations, which indicates that infrequent classes (which are the ones that get most attention from macroaveraged measures) need all bits of available information in order to be correctly handled.

A second observation is that some individual feature types deliver reasonably good accuracy by themselves. While some of them produce good results for F_1^μ only ("AppleClassScheme" and "GoogleClassScheme"), others ("Name" and "Description") provide fairly good values for both F_1^M and F_1^μ. "AppleClassScheme" and "GoogleClassScheme" provide information about the class assigned to the app in the corresponding stores; this is clearly very useful semantic information, especially for very popular classes (this should explain the very good micro accuracy obtained in the results). "Name" features also perform well, and this is intuitive, since authors tend to use highly significant words in app names. That "Description" features are helpful is unsurprising too, since the content of this field gives a very detailed information about the aim and functionality of the application, i.e., it is the field where content is meant to be conveyed.

We do not include figures for the remaining feature types ("AverageUserRating", "UserRatingCount" and "FileSize") since, when used in isolation of the others, they turn out to have no discriminatory power at all, since each of the three classifiers generated out of them is the majority class classifier (i.e., it always assigns Games and never assigns any of the other classes).

[13]The three features **AverageUserRating**, **UserRatingCount**, and **FileSize**, are exceptions, since Equation 1 discriminates between presence and absence of the feature in the app, and is thus unsuitable for features that have a quintessentially numerical nature such as these. Since these features are only three, we add them to the set of selected features without subjecting them to the selection process.

4.2 Effects of Feature Selection

The results show that feature selection can globally improve the accuracy of the classifiers on the infrequent classes.

The "Description" feature-specific classifier benefits from feature selection. This feature type contains the majority of features extracted from the dataset (45,652 out of 50,371), so a reduction in the number of features is necessary to prevent the problem of data overfitting.

For the "Name" feature type we observe a substantial cut, due to the fact that many app names have an evocative rather than a descriptive nature, and only few of them contain terms related to the functionality of the application. Some examples of useful app names, in terms of features, are "Google Calendar", "Jorte Calendar", "Aviary Photo Editor", "Photo Collage Editor". After the cut, app with fancy names cannot be classified with this only feature type, for this reason we see a large drop in F_1^M. The features related to store classification schemes receive a smaller cut than for "Names" features, but due to the uniqueness of these features (only one classification scheme feature per app exists), some apps are not more represented by these features after feature selection. We thus see the same drop in effectiveness. This happens because apps with store classes which are not discriminating for our classification scheme (i.e., classes which do not have a corresponding concept in our classification scheme), cannot be classified because these features are cut with high probability.

5. CONCLUSIONS

We have developed a system for the automatic classification of mobile apps by genre, and we have evaluated experimentally its effectiveness. We have found that the metadata retrieved from app stores is of fundamental importance in representing the content of apps, but also that classification via supervised learning is a difficult task. Accuracy is still less than optimal; there are still large margins of improvement. In the future we plan to explore further directions, among which (i) adopting learning algorithms specifically devised for extremely unbalanced datasets, (ii) exploring the use of hierarchical classification schemes, and (iii) testing the use of methods for the (semi-)automatic generation of additional training examples.

6. REFERENCES

[1] A. Finkelstein, M. Harman, Y. Jia, F. Sarro, and Y. Zhang. Mining app stores: Extracting technical, business and customer rating information for analysis and prediction. Technical Report RN/13/21, Department of Computer Sciences, University College London, London, UK, 2013.

[2] G. Forman. A pitfall and solution in multi-class feature selection for text classification. Proceedings of ICML 2004, pages 38–45, Banff, CA, 2004.

[3] T. Joachims. Training linear SVMs in linear time. Proceedings of KDD 2006, pages 217–226, Philadelphia, US, 2006.

[4] S. Robertson. Understanding inverse document frequency: On theoretical arguments for IDF. *Journal of Documentation*, 60(5):503–520, 2004.

[5] H. Zhu, E. Chen, H. Xiong, H. Cao, and J. Tian. Mobile app classification with enriched contextual information. *IEEE Transactions on Mobile Computing*, 13(7):1550–1563, 2014.