**Code language detection** The main goal of this project is to find out how methods of Cooperative Game Theory can be applied to code language detection.

**Model** Let's define a weighted voting game $\Gamma = \langle q; w_1, w_2, ..., w_n \rangle$ for code snippet, where $w_i$ is count of $i$ symbol from set

```
!\"#$%&'()*+,-./:;<=>?@[\]^_‘{|}~
```

and $q = \sum w_i \cdot q_p$ - quote, required for coalition to win. Value $q_p = (0, 1)$ is a voting threshold (in this paper $q_p = 0.75$). Assuming $q$ and $w_i$ positive integers, $0 < w_i < q$. Characteristic function of coalition $S$ defined as $v(S) = 1$ if $\sum_{w_i \in S} \geq q, 0$ otherwise.

For each code snippet we can calculate a Banzhaf power index $\beta(v) = (\beta_1(v), ..., \beta_n(v))$, which will represents power of each player (symbol) in the text. Result vector can be used as text/language fingerprint.

**Training data.** This method is very sensitive of training data quality. The best way is to review all input manually, but for this model data will be generated using Github Search API. For common languages it returns relevant code, but for some non wide used (Logo, Raku) or for languages that are sub/over sets of others (Gradle, modifications of Paskal, Lisp) it can return invalid data. In this paper 30 snippets of each language will be used as training and validation data.

**Pre-analysis.** At first we need to check that $\beta(v)$ vectors from proposed method represents difference between languages. On Figure 1 we can see that vectors distribution calculated for single language have stable pattern. But if we review values of $\beta(v)$ for all languages, there is no such pattern (Figure 2).

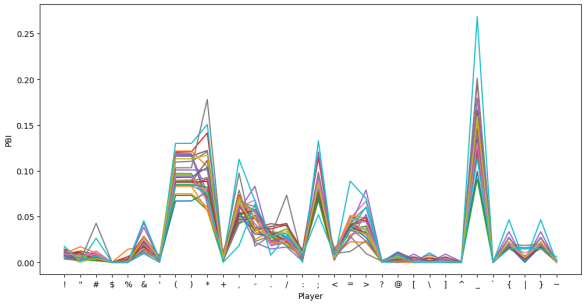


Figure 1: Values of $\beta(v)$ for C language

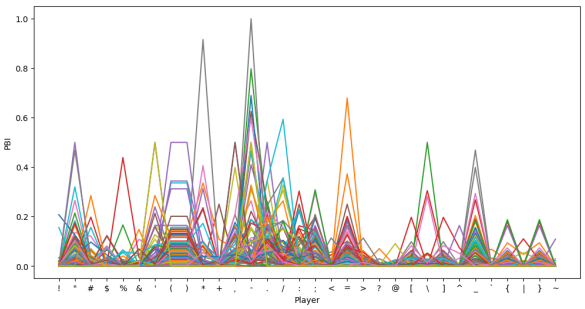


Figure 2: Values of $\beta(v)$ for all languages

**Experiments.** For method validation we will use simple algorithm: find nearest $\beta(v)$ from training data to target text $\beta(v)$ using Euclidean distance ($d$). Taking in mind that training data might contain invalid code snippets, additional condition will be added to smooth validation results: find average $\sum d$ for each language and if it's not equal to first step result, use average result.

**Results.** We will review 3 metrics: $R_1$ - exact match using just Euclidean distance, $R_{avr}$ - match considering average distance, $R_3$ - is language matched in 3 lowest distances. Results are shown in Table 1, $S(t, v)$ represents success rate for $t$ training code snippets and $v$ validation snippets.

Table 1: Experiments results

| Metric | $S(10, 20)$ | $S(15, 15)$ | $S(20, 10)$ |
|---|---|---|---|
| $R_1$ | 44% | 48% | 52% |
| $R_{avr}$ | 49% | 51% | 57% |
| $R_3$ | 61% | 66% | 71% |

In $S(10, 20)$ for languages C, CMake, GraphQL, INI, JSON, SmallTalk $R_1$ success rate is over 90%. For 25 languages success rate is over 60%.

**Final version** For a final submission Linear Support Vector Classification used for train and prediction. Final scores for $R_1$ are presented in Table 2.

Table 2: Linear SVC results

| Metric | $S(10, 20)$ | $S(15, 15)$ | $S(20, 10)$ |
|---|---|---|---|
| $R_1$ | 49% | 54% | 56% |

Increasing number of train data and verifying it may lead to better results. Most expensive calculation in algorithm is finding $\beta(v)$, but calculation time is not depends on any parameter, except initial number of players in game $|\Gamma| = 32$, memory consumption only depends on initial file parsing implementation.