Code Flow

1. Preprocessing

First, we use `panda` library to form our dataframes:

```
train_df = pd.read_csv(BOOKS_TRAIN_PATH)
test_df = pd.read_csv(BOOKS_TEST_PATH)
```

Then, we preprocess on our dataframes using `panda` library's built in functions and `hazm` library's `normalize` and `tokenize` functions to convert our text to an array of words.

```
def normalize_text(text):
    text = norm.normalize(text)
    return tok.tokenize(text)

def preprocess_df(dataframe):
    dataframe.apply(normalize_row, axis=1)
```

1. Creating Bag of Words

We form our BoW which is a 6 x **num_of_words** dict which: `bow[c][w] = number of times word: w appeard in books with category: c`

```
bow = dict()
for c in CATEGORIES:
    bow[c] = dict()

for _, book in dataframe.iterrows():
    for word in book.title:
        if not word in bow[CATEGORIES[0]]:
            for c in CATEGORIES:
                bow[c][word] = 0
        bow[book.categories][word] += 1

    for word in book.description:
        if not word in bow[CATEGORIES[0]]:
            for c in CATEGORIES:
                bow[c][word] = 0
        bow[book.categories][word] += 1

return bow
```

3. Prediction

I defined 3 functions here:

1. `prob_word_if_cat(bow, word, category, dot_product)`: Probability of having `word` on `category`. If `word` is not present in `bow` or `category`, Additive Smoothing rule is applied (with **alpha** = 1).
2. `prob_cat_if_book(bow, book, category, dot_product)`: Probability of `book` being in `category`. Calculated using Bayes theorem.
3. `predict_cat(test_df, bow)`: Runs a loop over all books in `test_df` and calculates probability for each book being on each category and chooses the category with maximum probability as the answer.

Note: `dot_product` is (as you might have guessed), the dot product of `bow` and a `1 x n` matrix full of ones.

Another Note: Since `P(C)` for every category is 1/6 (because the number of all books in each category is equal), we can ignore that in our summation. Also, we can return the summation result in function #2 because `e^x` and `x` are both ascending functions.

Optimizations

First of all, the program is slow (I mean, really, really slow). It takes 20 seconds to run (this is almost the most accurate run with all optimizations on).

```
❯ python3 src/main.py

Reading CSV: 0.1304283059998852
Preprocessing: 11.857562787000006
Creating BoW: 3.8803586969997923
Prediction: 0.44412514200030273
Accuracy:  82.66666666666667%
```

---

1. Preprocessing Optimizations

1. We can remove stop words (conjuctions, numeric words and `hazm`'s `stopwords_list` from our BoW:

```
def filter_row(row):
    row.title = list(filter(is_important, row.title))
    row.description = list(filter(is_important, row.description))
    return row

def is_important(word):
    if re.search(r'\d', word): return False
    if word in CONJUCTIONS: return False
    if word in STOP_WORDS: return False
    return True
```

2. We can also lemmitize our words using `hazm`'s `Lemmatizer`:

```python
def lemmatize_row(row):
    row.title = list(map(clean_word, row.title))
    row.description = list(map(clean_word, row.description))
    return row

def clean_word(word):
    word = lem.lemmatize(word).split("#")[-1]
    return word

def preprocess_df(dataframe):
    dataframe.apply(normalize_row, axis=1)
    dataframe.apply(filter_row, axis=1)
    dataframe.apply(lemmatize_row, axis=1)
```

Now we have and array of roots of each word in persian, so words from same root with different shapes become the same and easier to proccess on.

BoW Optimizations

1. Let's have a guess: words that in title are more important than words in description. So what if we give them weights?

```python
for word in book.title:
    if not word in bow[CATEGORIES[0]]:
        for c in CATEGORIES:
            bow[c][word] = 0
    bow[book.categories][word] += WEIGHT

for word in book.description:
    if not word in bow[CATEGORIES[0]]:
        for c in CATEGORIES:
            bow[c][word] = 0
    bow[book.categories][word] += 1
```

`WEIGHT`	1	5	10	100
Accuracy	81.7%	82.6%	82.6%	79.1%

Yay! It seems like `WEIGHT = 5` is a good one.

Prediction Optimizations

1. Using additive smoothing:

```python
def prob_word_if_cat(bow, word, category, dot_product):
    if word in bow[CATEGORIES[0]]:
        n_w = bow[category][word]
        if n_w == 0:
            return ALPHA / (dot_product[category] + ALPHA * len(bow[CATEGORIES[0]]))
        return n_w / dot_product[category]
```

```
    else:
        return ALPHA / (dot_product[category] + ALPHA * len(bow[CATEGORIES[0]]))
```

Now let's see the result with different `ALPHA` values:

Alpha	0.01	0.1	1	10	100
Accuracy (%)	76.2%	78.2%	82.6%	79.3%	78.2%

Results:

With Additive Smoothing (ALPHA = 1):

-	Removing Stop Words	Keeping Stop Words
Lemmatize	16s, 82.6%	20s, 71.5%
No Lemmatize	15s, 78.8%	16s, 73.3%

Without Additive Smoothing:

-	Removing Stop Words	Keeping Stop Words
Lemmatize	16s, 4.2%	18s, 7.5%
No Lemmatize	15s, 0.8%	16s, 1.3%

Why additive smoothing is so effective?

First of all, our data is very limited. only ~26k words are present in BoW and that is all we got. When a new word appears in test cases, ignoring it wouldn't be logical. The additive smoothing technique helps us to calculate a prediction for that situation, and also it **smooth**es the probability distribution graph.

Total Result:

Gussed ->	مدیریت و کسب و کار	رمان	کلیات اسلام	داستان کودک و نوجوانان	جامعه شناسی	داستان کوتاه	Accuracy (%)
مدیریت کسب و کار	69	0	0	0	6	0	92%
رمان	1	53	1	2	1	17	70%
کلیات اسلام	0	0	62	3	9	1	82%
داستان کودک و نوجوانان	1	4	2	64	0	4	85%

Gussed ->	مدیریت و کسب و کار	رمان	کلیات اسلام	داستان کودک و نوجوانان	جامعه‌شناسی	داستان کوتاه	**Accuracy (%)**
جامعه‌شناسی	5	1	2	1	65	1	86%
داستان کوتاه	1	8	1	6	1	58	77%
Total Accuracy	-	-	-	-	-	-	**82.6%**