Project Phoenix: VOTE in the 21st Century

William Edward Bailey, III

April 18th, 2016

Advisor: Stephen Slade
Dept. of Computer Science

I. Background

During the late 80s early 90s, Professor Slade wrote the automated decision system known as VOTE. The goal of VOTE was to apply the concepts of qualitative goal-based decision-making to the domain of politics. Members of Congress make decisions on bills on a weekly if not daily basis. What if there were a way to predict how each member of Congress would vote and then use this data to predict how likely the bill would pass? What if a rationale could be given for these decisions so that others would know what factors could sway policymakers in one way or another? VOTE was designed to tackle these problems.

As a final project for CPSC 558 Automated Decision Systems, Krishnan Srinivasan and I worked on translating VOTE from Lisp into Python. Our project was divided into three main phases. First, we translated the core code of VOTE stored in the /pol folder. Second, we configured a Mongo database to store our objects. Third, we proofed our code to produce a rudimentary working version. At the end of the project, VOTE possessed three strategies with which it could make decisions for a few members of Congress on a few bills.

The goal of this project has been to continue the work from last semester and refine the VOTE system. The sections below cover the objectives in more detail. The first section provides
a general overview of VOTE and how the code works. The second section discusses the work accomplished over this semester and how the deliverables specified in the proposal were fulfilled. The third summarizes other key improvements I made to the code. The last section describes ways the system could be extended in the future.

II. VOTE Overview

A. Overview of the File Structure

In the first implementation of VOTE, all the code was stored under a single folder called /pol. The new code is organized into several different packages. The code was designed to be modular with clearly defined functionality.

- /database = Contains the json files for the database
- /test = Contains all the test code for the project
- /log = Contains log data for vote
- /src = Source code
  - /src/analyze = Contains classes for computing metrics on members or decisions
  - /src/classes = Contains classes that define objects in VOTE like Member
    - /src/classes/data = Helper classes that encapsulate data like the importance of a bill
    - /src/classes/strategies = Contains the code for strategies
  - /src/config = Contains scripts and files to set up the configurations for running
VOTE

- `/src/constants` = Constants that are used throughout the code
- `/src/database` = Defines the interface for the database
- `/src/scripts` = Contains scripts such as loading the database or configuring logging
- `/src/util` = Various utility functions
- `/src/vote` = The main code of vote

B. Overview of the Code Structure

VOTE follows an object-oriented approach. The `/src/classes` folder contains the main classes that VOTE uses in making decisions. Member objects represent given members of Congress. In addition to basic biographical information, members have a `voting_record`, `credo`, and `relations` (groups they are affiliated with). Groups are defined in `group.py` and represent interest groups or political parties like Republicans or Democrats. A Relation object from `relation.py` describes the relationship a member has with a given group (whether it is positive or negative, how important it is, etc.). Bill objects represent the bills themselves. In particular, each bill has `stances_for` and `stances_agn`, which contain stances on issues that can be inferred from voting a certain way on this bill. For example, voting for a commerce bill would show support for the economy. Stance objects are defined in `stance.py`; Issue objects are further defined in `issue.py`.

The Decision object is the heart of VOTE. Defined in `decision.py`, it contains important metrics that are used to calculate a member's decision on a bill. Many of these fields are populated by code that can be found under `/src/analyze`. For example, `member_analyze.py`
defines two important methods. `extract_voting_stances` determines what stances a member might hold based on his/her voting history. `infer_relations_stances` infers what stances a member may hold given the relationships that he/she has with groups. `decision_analyze.py` uses these stances to produce metrics such as the norms on the issues being considered, whether or not the member's credo is split (the credo has stances that would support voting both for and against the bill), and whether or not a given source of stances (voting record, credo, relations) has a general consensus to vote for or against the bill.

These class are utilized by `/src/vote/vote.py`. In particular, the file defines the methods `vote` and `vote_all` which simulate votes of specific members on particular bills. They first initializes a decision object with useful metrics using the `/src/analyze` code above. After that, they apply Strategy objects to the decision objects. Strategies are defined under `/src/classes/strategies` and provide various heuristics for making a decision. For example, the `NotConstitutionalStrategy` first checks to see if a general consensus has been reached on the bill. In other words, it checks whether the member's credo, voting record, groups he/she cares about, etc. are overall in favor of voting the same way on the bill. Furthermore, if one reason to vote against the bill is that it is unconstitutional, the `NotConstitutionalStrategy` will conclude that the member will vote against the bill. Hence, `vote.py` will apply numerous strategies until either one succeeds or they all fail. It will then report the results back to the user.
III. Deliverables

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Files</td>
<td>49</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>1342</td>
</tr>
<tr>
<td>Lines of Comments</td>
<td>2156</td>
</tr>
<tr>
<td>Tests (Total)</td>
<td>310</td>
</tr>
<tr>
<td>Unit Tests</td>
<td>283</td>
</tr>
<tr>
<td>Staging Tests</td>
<td>27</td>
</tr>
<tr>
<td>Test Files</td>
<td>40</td>
</tr>
<tr>
<td>Lines of Test Code</td>
<td>3549</td>
</tr>
<tr>
<td>Lines of Test Comments</td>
<td>1105</td>
</tr>
</tbody>
</table>

Note: Files include __init__.py. Comments include the license at the top of each file.

A. Intro

In my proposal, I stated that I would (1) refine the remaining strategies and (2) configure vote to pull data from a website. As I began refining the strategies, it became apparent that more substantial changes were needed. I first started by refining the strategies under the existing /pol structure. My refinement of the Non-partisan Strategy is a good example of my realization that my approach needed to change. I had translated the code and got a sample working for a Republican member. However, I had a bug in the code that handled Democratic members. I happened to run across this later while working on something unrelated. At the time, all my code was stored under /pol and there were no test cases. While I could continue working on strategies, the foundation was shaky at best. Hence, I decided that I should focus on reworking the existing code and adding tests.

As mentioned, the first deliverable was to refine the remaining strategies. I changed this to be (1) refining all of my existing code that is necessary for VOTE to make a decision on a bill.
In addition to what is described in part B, this also includes some of the changes mentioned in the Key Improvements section below. As for the strategies themselves, I refined 15 out of 18 strategies making sure that each had a suite of unit tests and staging tests.

The second deliverable was to populate the database with real data. As mentioned earlier, I realized that doing one of the additional improvements (namely the Test Suite) was more urgent. Real data is no good without a solid foundation. Hence, I decided to develop a test suite and good comments in lieu of getting real data. The goal was to provide a robust and well-documented system that could be easily extended later and would not have errors such as the one mentioned at the beginning.

**B. Refining of Core Code**

Since I had worked on VOTE last semester, I had a better idea of how all the pieces of code fit together. Hence, I could write the code in a more organized and Pythonic fashion. As mentioned earlier, a major restructuring involved dividing the `/pol` code to the different packages I now have. For `/src/classes` objects, I spent some time removing attributes that were no longer needed and providing more descriptive names to the ones that were left. For example, here is the initialization for variables in a Member object:

```
/pol code:

self.name = None
self.fname = None
self.lname = None
self.english_short = None
self.notes = []
self.gender = None
self.votes = []
self.new_votes = []
self.stances = []
```
Other times, the changes were more organizational in nature. In the /pol code, methods that were used to determine decision metrics were scattered in different places. Some like `match_stances` and `group_reasons` were held in `vote.py`. Others were held in `decision_stats.py` (the predecessor to `decision_analyze.py`). Although `decision_stats.py` contained code relating specifically to decision metrics, it also contained more generic methods like `collect_bills`, which
was used to collect stances that came from bills. Lastly, *member_stats.py* held some metric codes for members. In the restructuring of the code, I organized the existing code into *decision_analyze.py*, *member_analyze.py*, *stance_analyze.py*, and *util.py*. The code in *decision_analyze* and *member_analyze* contained only the code directly related to computing decision and member metrics specifically. As a result, *match_stances* from *vote.py* was moved into *member_analyze*. For generic methods like *collect_bills*, I moved them into *util.py*, which provided generic helper methods for filtering or modifying lists. *stance_analyze.py* took over methods that required substantive computation on stances like *group_reasons* or *normative_stance* (queries the database to see if the stance is normative). With this more modular structure, it will be easier to modify and update code moving forward.

Sometimes, the changes required entirely new approaches to the code. Handling the strategy code is a prime example. Under the old system (*/pol*), Strategy objects were stored in the database. Among other things, these objects specified (1) the name of the strategy, (2) the name of the method that implemented the strategy code, and (3) the name of a function that explained the strategy's decision. The logic for all strategies was implemented under *strategies.py* while the explanation functions were all implemented under *protocol.py*. When *vote.py* would apply strategies, it would query the database for the Strategy objects. It would then look up the name of the strategy code in its global variable table and run the code that was there. It would then look up the function for an explanation in a similar fashion.

This approach had massive security concerns. A user could provide the name of any function not just strategy functions. If the function was in the global table, it would be run. To change this, I created a new structure under *src/classes/strategies*. Like the old Strategy objects,
the new StrategyEntry object (strategy_entry.py) specifies the name of the strategy, whether the
strategy is active (should be run), and the rank of the strategy (what order it should be run).
However, the name of the strategy indexes into a hash under strategy_hash.py. This hash maps
strategy names to the classes that implement the functionality for these strategies. Hence, if a
strategy name from the database is not in this hash, an error will be logged and no harm will be
done.

As mentioned earlier, the code for all strategies was listed under /pol/strategies.py while
all the explanation functions were under /pol/protocol.py. I used an object-oriented approach to
fuse these two files. My new Strategy object under /src/classes/strategies/strategy.py is an
abstract class that is the parent of all strategies. It defines generic code that all strategies will
have like run, explain, consensus, and majority. It then provides a private _run and _explain
method that it expects child strategies to override. Each child strategy like
best_for_the_country_strategy.py overrides _run() with its unique strategy code (once stored in
/pol/strategies.py) and then overrides _explain() with the unique explanation code (once stored in
/pol/protocol.py). As a result, all code related to a specific strategy is stored in one place while
code for different strategies are stored in separate files.

Lastly, it is important to note how I simplified code to be more Pythonic. To demonstrate
this, I will show the code for the Non-partisan Strategy under both systems.

/pol code:

```python
def strat_non_partisan(decision, strat):
    member = get(MEMBER, {"_id" : decision.member})

    credo = decision.MI_credo

    if credo in [None, []]:
        return None
```
```python
credo_side = credo[0]
credo_stance_list = credo[1]

opposing_groups = decision.group_agn if credo_side == FOR else decision.group_for

party = UNKNOWN_PARTY
if member.party == REP:
    party = REPUBLICANS
if member.party == DEM:
    party = DEMOCRATS

credo_stance1 = credo_stance_list[0]

mapfun = lambda stance: stance.source

if (credo and opposing_groups and credo_stance_list and
    party in map(mapfun, opposing_groups) and
    most_important(credo_stance1.importance)):
    return set_decision_outcome(decision, credo_side, strat)
else:
    return None
```

/src code:

```python
def _run(self):
    
    """Implements the logic of Non-partisan Decision. Simply put, if one strongly cares about one's credo and one's party is against the credo, vote in line with one's convictions.

    To be more specific, stances in the member's credo must be of importance A, the highest possible.

    If defined, decision.MI_*** contain stances all of the same importance.
    Please see decision_analyze.py for more details.
    """
    credo = self._decision.MI_credo
    party = self._member.party
    self._opposing_groups = self._decision.groups_agn
    if credo and credo.outcome == outcomes.AGN:
        self._opposing_groups = self._decision.groups_for

    if credo is None or party is None or not self._opposing_groups:
        return

    mapfun = lambda stance: stance.source
```
credo_stance = credo.data[0]
if (party in map(mapfun, self._opposing_groups) and
credo_stance.importance.most_important()):
    self._set_decision(credo.outcome)

As can be seen, some of the structural changes I made enabled me to write code more clearly. By initializing each strategy with the member that is voting, I no longer need to query the DB each time to get the Member object but can do self._member. I created a ResultData object under /src/classes/data/result_data.py which provides two attributes outcome and data. This is useful for storing the results of an MI_stance. Previously the stances were stored as an array in the form [outcome_such_as_FOR_or_AGN, [list_of_stances_supporting_outcome]]. Hence, the old code had to access these values with magic number indices. By using the ResultData object, the outcome of the MI_credo can be retrieved under the outcome attribute and the stances that support this decision can be found under the data attribute. The code is more generic by not assuming that there are only two parties; hence, new parties such as Independent can be added to the database and the strategy will still work. Lastly, checking whether variables are None is done at the top of the function so that the latter if statement is easier to read.

C. Testing

As mentioned, a robust test suite is needed to weed out errors and ensure the foundation of VOTE is solid. All tests are stored under the /test folder. It's structure mirrors /src containing test cases for the corresponding files. I have two types of tests: unit tests that are responsible for individual files and staging tests that tests whether VOTE produces the right decision for a given set of data. Currently, I have 310 tests (283 unit tests and 27 staging tests) that provide coverage over the different packages.
For the unit tests, I write numerous tests to ensure the various control flow paths work as expected. As a sample of how I test a given file, I have included an example below from /test/classes/strategies/not_constitutional_strategy_test.py:

```python
def test_init(self):
    """Tests whether a strategy can be constructed properly""

def test_run_fail_no_constitutional_issue_in_db(self):
    """ Verifies the function fails if there is no Constitution issue.""

def test_run_fail_no_consensus(self):
    """ Verifies the function fails if there is no consensus.""

    def test_run_fail_consensus_for(self):
        """ Verifies the function fails if there is a consensus FOR the bill.""

    def test_run_fail_no_constitutional_stance_in_agn_stances(self):
        """ Verifies the function fails if the bill isn't unconstitutional.""

    def test_run_success(self):
        """ Verifies that run() successfully makes a decision""

    def test_explain(self):
        """ Verifies explain runs if there is a success [aka _explain is implemented].""
```

While I often mock data, I tend not to mock methods between modules. By doing so, the unit tests ensures that the code calls the interfaces it relies on properly. I don't mock out the MongoDB either but run the tests with a test database to ensure that the code works properly with the DB.

For the staging tests, I provided sample data (stored in the json under /database). I then called vote on a member and a given bill and made sure that the appropriate strategy was used to
make the decision. This tested the end-to-end flow of VOTE to ensure that the interfaces between different code worked properly.

D. Commenting

Opaque code is inflexible code. If the intention of the code is not clear, it is difficult to extend. If the interactions between modules are not specified, errors are bound to occur when modifications are made. Hence, I made an effort to extensively comment the code. I built upon the comments from last semester either adding to them or rewriting them entirely. In the /src folder, I have 2156 lines of comments. Within /test, I have 1105 comments. In addition to including a license notice at the top of each file, the files in /src describe what the code does and how it should be used. I have provided several examples below:

/src/classes/stance.py

class Stance(PrintableObject):

    """Represents what an entity feels about a given issue.

    Stances are key to VOTE. In short, a stance tells what it's owner feels about a given issue. It defines how strongly the owner feels about the issue (importance), what side the owner is on (PRO, CON), etc. Typically, members and groups hold stances. Bills also contain stances denoting how voting for or against the bill will reflect what issues a member supports or disapproves of.

    Attributes:
    source: An identifier for the source of the relationship. It could be an id, name, synonym, etc.
    source_db: The name of the database table that source belongs to
    issue: An identifier of the issue the stance is on
    importance: An _Importance object denoting how important the stance is to the source
    side: Whether the source is for (PRO) or against (CON) the issue
    relation: An optional argument. If this stance was adopted from a relationship, this entry holds the relationship it was adopted
from siblings: A list of related stances. The stance_alike(...) method must return true in order for a stance to be in this list
sort_key: How stances should be sorted. This is typically set by the owner of the stance

/src/analyze/decision_analyze.py
def _compare_stances(fors, agns):
    """This function compares the stances to see which side has the most compelling reasons. It does so by checking which side has the largest number and most important stances supporting it.

    Arguments:
    fors: the list of stances supporting the bill to decide on
    agns: the list of stances against the bill to decide on

    Returns:
    A list containing the strongest arguments for or against the bill. Returns an empty list if both sides are equally compelling.

    Notes:
    Both lists are first sorted by importance. Then, it goes through and compares each stance.

    If one list runs out before the other, the longest list is considered to have the most compelling stances.

    If one list is found to have a stance of stronger importance than the other on a given iteration of the for loop, that list is considered to be the most important and is chosen.
    """

/src/classes/strategies/strategy.py
class Strategy(PrintableObject):
    """""""This is the parent class for all strategies. It defines an interface that all subclasses are expected to implement and abide by. Subclasses will
implement the specific strategy logic.

A Strategy provides a means for determining how a member will vote on a given bill. It takes in a Decision object and then uses the statistics within the object to determine how the member will vote. It applies specific heuristics to calculate this result. If successful, the Strategy records the predicted vote and the reason for this vote in the Decision object it is passed.

Child classes should override the _run() and _explain() methods. They should also override __init__(...) to specify there own name. However, they are expected to keep the signature of __init__(...) exactly the same.

Attributes:
    _name: The string name of this strategy
    _decision: The Decision object representing the decision the strategy is trying to compute
    _member: A Member object that represents the member who is making the decision
    _bill: A bill object that represents the bill being voted upon.
    _success: Whether or not this Strategy was successful in computing a result for the Decision object

```
/src/vote/vote.py

def vote_all(member_identifier=None, bill_identifier=None):
    """Run the vote program for all members on all bills. The function has optional parameters to run on specific members or bills. For example, by specifying member_identifier, the function can simulate all decisions that particular member would make on all bills. By specifying bill_identifier, the function can simulate the decisions that all members would make on that particular bill. If the arguments are left as None, all members and all bills will be used.

Arguments:
    member_identifier: optional, an identifier such as full_name of the member to simulate
    bill_identifier: optional, the bill identifier of the bill to decide on
```

Usage Examples:
vote_all() : process all members on all bills
vote_all(member_name = name) : process all bills for given member
vote_all(bill_name = bill) : process all members for a given bill

IV. Key Improvements

A. Unified Database

The Python version of VOTE utilizes MongoDB to store the different objects under /src/classes (members, bills, issues, groups, strategy_entries, and decisions). The data is initially stored as json under the /database folder. These json files are loaded into the database via /src/scripts/database/load_data.py.

In the first implementation of VOTE, the database was divided. We had two sets of json files: one to define members, groups, etc. and another to define the stances and relations. We would first load the basic objects into the database. Then, we would use the stance/relation json files to define the stances and relations for the members and groups.

In order to unify the database, I first had to properly create a transformation class. The goal was to enable objects from /src/classes to be passed directly to the database. The database would then store these objects. When an object needed to be retrieved, the database would return the object. Simply put, one would pass in a Member object and receive a Member object. However, Mongo does not know how to handle custom objects: it only knows how to handle dictionaries. Hence, I rewrote /src/database/vote_transform.py to be able to translate a custom object into a dictionary and then later convert the dictionary back into an object. It starts by recursively looking through a dictionary. During encoding, it looks for custom objects. When it finds one, it encodes the __dict__ of the object and then adds a key “_type” that denotes the type
of the object. It then replaces the custom object with this dictionary. Upon decoding, it looks for
dictionaries with the “_type” key. It then creates a new object of the appropriate type.

In order to apply this translation, I needed to create a wrapper for Mongo. This is the job
of /src/database/pymogodb.py. It defines the PymongoDB class which implements the basic find,
insertion, and replacement commands. It encodes data heading into the database and decodes
data leaving the database. It calls the pymongo module to actually store data in Mongo. Since the
translation works recursively, a member along with its stances and relations can be stored in one
json file. Hence, /src/scripts/database/load_data.py and /src/scripts/database/dump_data.py load
json files to the database and dump the data to json respectively.

B. Config Files

Ideally, vote would be run under different settings: production, development, staging, and
testing. Each of these areas would require different setup and each would need its own database.
This is the goal of /src/config/config.py. It reads in a specified config file and stores the results in
the CONFIG hash. The config files are also stored under /src/config and contain parameters like
the DATABASE to use (prod vs. test). Before VOTE is run, the proper config file is loaded.
When the database module of /src/database is setting up the database, it uses the values from
CONFIG to choose which database to access (prod vs. test).

C. Logging

Initially, the code recorded output by printing to the screen. To provide a more robust
system, I set up logging. In /src/scripts/configure_logging.py, the root logger is set up properly
based on CONFIG parameters. If LOG is enabled, the logger will write to a file. The file will be
of LOG_SIZE. Once the log is full, it will then create a new log and save the old as a backup. It
will have at most LOG_BACKUP_SIZE of backup files at any time. Once that is filled, it will
erase the oldest log and create a new one (so as not to fill the system with too many logs). If
DEBUG is enabled, logging will also print to the screen. If it is disabled, only severe errors will
be recorded. Log entries follow the format:

[log_severity  time @ file:method:line_number] message

For example:

[INFO 2016-03-30 14:29:13,521 @ src/analyze/member_analyze.py:extract_voting_stances:40]
Extracting stances based on voting record of Stan Parris...

V. Future Work

A. Finish Strategies

I currently have implemented 15 Strategies with unit and staging tests. There remains 3
strategies to be implemented (Partisan, Shifting Alliance, and Deeper Analysis). Hence, I could
refine these final strategies and implement tests for them. After these were done, all the old
strategies of VOTE would be implemented. I could then focus on determining what is the best
order to apply the strategies, as multiple strategies might succeed in a given case. Which strategy
VOTE chooses to make a decision will influence what explanation is produced.
B. VOTE for Bills

Currently, VOTE is focused on determining how a member will VOTE on a bill. It does not provide a direct way to determine whether a bill would pass or fail. While one could run VOTE for each member, the user would be responsible for tallying the results. Hence, a new vote function could be implemented to determine whether or not a bill would pass. It would run all members for a given bill, tally the results, and return them to the user. This would allow for an interesting form of feedback. One Strategy (Could Not Pass) makes a decision based on how many votes there are FOR or AGN a bill. If the bill has too many AGN votes, the member may vote against the bill if there is a majority against it. As VOTE runs for more members (aka members have more information about how Congress as a whole feels on the bill), the decisions the members make might change. Hence, an iterative scheme could be applied till the decision on the bill converged.

C. Live Data

With the engine of VOTE finished, the data pipeline could then be implemented. The goal would be to scrape data from various sources to populate the databases. However, this would require a rather nuanced approach. The pipeline would need to determine what issues are important to members and how important they are. The pipeline would have to determine the member's friends (and enemies) and determine how much the member cares for (or dislikes) them. Much of this data is qualitative and hence will rely heavily on semantic processing of texts. Since good qualitative data is important for VOTE, it is important to develop tools that will produce models that accurately portray the concerns of the member.
D. Website

A website would be the best way to publicize VOTE. One could create the website using Ruby-on-Rails. The frontend would query the database to find the list of members and bills. It would then provide an interface for users to select which member and bill to make a decision on. The frontend will then call the backend, which will run the VOTE program. The backend will receive a decision object, which it can then send to the frontend. The frontend can use this object to display the results to the user. If the frontend also wished to display the reasoning behind the decision, the results from Strategy.explain() would have to be captured and sent to the frontend as well.

VI. Conclusion

The objectives of this project have changed for the better from the start of the semester. Initially, I planned to refine the strategies and fill the database with real data. I changed this to refactoring the entire code to make it (1) modular, (2) more extensively documented, and (3) heavily tested. 310 tests ensure that 1342 lines of code work properly. 2156 lines of comments describe how this code should be used and extended. Currently, the system is equipped to predict decisions on bills with 15 different strategies. VOTE has in place a solid infrastructure that can support real data and other future extensions like a website. All code for this semester and last semester is available at: https://github.com/WEB3-GForce/VOTE. I have enjoyed working on VOTE and look forward to seeing how it might grow in the years to come.