Online Appendix for "How Do You Say Your Name? Difficult-To-Pronounce Names and Labor Market Outcomes"

Qi Ge* Stephen Wu†

Abstract: This online appendix contains additional empirical analyses complementing the results and discussions presented in the main text. In Appendix A, we perform robustness checks on our baseline findings using observational data from the academic labor market. In Appendix B, we explore the possibility that the uniqueness or commonality of names may affect job outcomes. In Appendix C, we test for heterogeneous effects by gender using experimental data from Bertrand and Mullainathan (2004) and Oreopoulos (2011). In Appendix D, we investigate labor market effects of name fluency using data from Nunley et al. (2015). Lastly, in Appendix E, we include the full instructions for our name fluency surveys.

*Department of Economics, Vassar College, Poughkeepsie, NY 12604, *qige@vassar.edu*.
†Department of Economics, Hamilton College, Clinton, NY 13323, *swu@hamilton.edu*.

Appendix A. Robustness Checks

In this section, we present a number of robustness checks on our baseline findings using observational data from the academic labor market. We first consider an alternative placement quality measure based on the raw RePEc ranking that excludes private sector and non-tenure track academic placements. Tobit estimates based on this alternative RePEc measure are reported in Table A2 in the Online Appendix and are similar in direction and statistical significance to the full sample results with imputed RePEc values as shown in Tables 2 and 3. Relative to the original imputed RePEc rankings, the relevant coefficients on name fluency measures are smaller in magnitude for the timing measure (51 vs. 83) and the subjective rating (28 vs. 82) but larger for the algorithmic rating (79 vs. 67).

Next, to assess the robustness of our specifications on placement quality, we estimate multinomial logit regressions for different types of placements and present the estimates in Table A3 in the Online Appendix. We observe that relative to the reference group placement type of government or think tank jobs, the coefficient on name difficulty is significantly negative for being placed into academia, and this result is consistent across different name fluency measures.[1] On the other hand, in separate specifications reported in Table A4 in the Online Appendix, when we further decompose academic job types and set the baseline category as visiting/postdoc, the coefficient on name difficulty for the tenure track category is not significant relative to the baseline. Taken together, this suggests that name fluency impacts the likelihood of being placed into academia relative to industry or government jobs, but does not affect the probability of obtaining a tenure track job, conditional on being placed in academia.

As an additional check on the robustness of the results on placement quality, we estimate an ordered probit model using categories of the imputed RePEc ranking of job placements as the outcome of interest. Given the ordinal nature of RePEc rankings, we categorize

[1] The difference between the coefficients for academic and industry positions is statistically significant at the 10%, 1%, and 1% levels for name fluency measures based on algorithmic ratings, pronunciation time, and subjective ratings, respectively.

the ranking of imputed RePEc productivity index into the following five categories for the ordered probit model: 1) RePEc ≤ 50; 2) $50 <$ RePEc ≤ 200; 3) $200 <$ RePEc ≤ 400; 4) $400 <$ RePEc ≤ 800; and 5) RePEc $= 1,000$. The estimates on name fluency measures, as presented in Table A5 in the Online Appendix, are qualitatively similar to our main findings and again suggest that candidates with harder-to-pronounce names tend to be placed in institutions with lower research productivity.

A concern discussed in the main text is that name changes may be endogenous. For example, students who have advisors and committee members from the same country might be less likely to feel the need to Americanize/Anglicize their (first) names. Ge et al. (2021) document a beneficial impact of student-graduate committee matching, in the form of country of origin and native language, on students' initial placement outcomes in the economics PhD job market, which could lead to a downward bias in the estimate of the magnitude of the name fluency effect. To account for this possibility, we re-estimate our baseline specifications and add controls for student-graduate committee matching based on country (U.S. vs. non-U.S.) or native language (English vs. other),[2] and the resulting estimates, as reported in Table A6, remain identical to those in Tables 2 and 3. The decision of whether or not to change one's last name after marriage may also be endogenous, though separate analysis by gender does not reveal any differences in the effects of name fluency. As shown in Table A7, we find similarly sized effects for the sample of male job market candidates (where changing last names is much less common than for females). Furthermore, as seen in Table A8, our results continue to hold when we exclude all candidates with ethnically Chinese names, a group for which individuals are particularly likely to adopt Americanized first names.

Another potential concern is that difficult-to-pronounce names are concentrated in a few countries, and the lack of success that individuals from these countries have in finding prestigious academic jobs is not necessarily linked to their names but from more general

[2] Following Ge et al. (2021), we code "country match" as being equal to one when at least one of the student's committee members went to an undergraduate institution in the same country as the student's undergraduate institution. Similarly, we code "language match" as being equal to one when a student's country of origin has the same official language as that of at least one of the committee members.

discrimination due to national origin. All regressions shown in our tables have controlled for the region of one's undergraduate school, but we have also estimated specifications which include a full set of individual country effects, and the results, as presented in Table A9 in the Online Appendix, are largely the same. In addition, we have also run separate regressions for different regions, though the statistical power is reduced in regions with few observations. In general, we observe that the effects of name fluency on placement types and quality are not driven by a particular region of undergraduate degree, as the magnitudes of the effects are large and significant for several different regions.

Appendix B. Common Names

We also explore the possibility that the uniqueness or commonality of names may affect job outcomes. It is likely that those with very common names could be at a disadvantage because they do not stand out from other candidates. Because pronunciation difficulty is likely negatively correlated with commonality of names, our estimates of the name fluency effect might be underestimated. To alleviate this concern, we augment our baseline specifications by controlling for having a common first name or common last name. Due to data constraints, we focus on common names in the U.S. Specifically, we code someone as having a very common name if their first name is among the 50 most common female first names or the 50 most common male first names according to the 1990 U.S. Census, and having a very common last name if their last name is among the 50 most common surnames according to the 2010 U.S. Census.[3]

We present the resulting estimates in Table A10 in the Online Appendix. As shown in columns 1-3 that focus on the full sample of job market candidates, none of the variables for name commonality (i.e., indicator for common first name, indicator for common last name, and their interaction) is statistically significant, and their inclusion does not impact

[3]The 1990 and 2010 U.S. Census data respectively represent the most recent data sources for tabulations on common first and last names.

the magnitude or significance of the name difficulty coefficient in any of our regressions. In addition, since the data sources for our common name analysis are based on the U.S. Census, we also conduct a separate analysis for the sample of job market candidates who are from U.S. and Canada. As shown in columns 4-6, the results on placement types and quality as well as the coefficients on common name indicators are qualitatively similar, though larger in magnitude.

Appendix C. Heterogeneous Effects by Gender in Audit Study Data

In this section, we explore potential gender differences in the name fluency effect in the experimental data from Bertrand and Mullainathan (2004) and Oreopoulos (2011). For each of these data sources, we divide the sample by gender and re-estimate our main probit regressions that relate callback rates to algorithmic ratings of name difficulty. All specifications include controls for name length, race/ethnicity, and resume characteristics and use standard errors that are clustered at the job advertisement level.

Table A15 in the Online Appendix reports our estimates for the name fluency effect by gender, with the top and bottom panels focusing on data from Bertrand and Mullainathan (2004) and Oreopoulos (2011), respectively. Columns 1-2 and 5-6 are based on the full sample of each data set, while columns 3-4 and 7-8 focus on the sample of Black job candidates and immigrants from India, Pakistan, and China, respectively. Although the point estimates on the name difficulty measure are somewhat larger and more statistically significant for female applicants across both data sets, the magnitudes of the impacts are not statistically different between the two groups.

In addition, we also compare the algorithmic ratings of names between male and female job applicants and find that there is no consistent and systematic relationship between fluency of names and gender across the two audit study data sources. Specifically, we find that female applicants, on average, have significantly more difficult (first) names than their male counterparts in Bertrand and Mullainathan (2004), while the opposite pattern holds for Oreopoulos (2011).

Overall, we do not find support for significant gender differences in the effect of name fluency based on prior audit study data. Our findings here also support the results in Table A7 in the Online Appendix that document indistinguishable name fluency effects between male and female economics PhD job market candidates.

Appendix D. Experimental Data from Nunley et al. (2015)

As an additional test, we also investigate labor market effects of name fluency using data from Nunley et al. (2015), who perform an audit study to examine racial discrimination in the labor market for recent college graduates. Specifically, Nunley et al. (2015) create fictitious and identical resumes for college-educated entry level job applicants who are randomly assigned one of the eight distinctively White-sounding or African American-sounding names. Similar to Bertrand and Mullainathan (2004) and Oreopoulos (2011), Nunley et al. (2015) also focus on callback rates as their main outcome variable of interest.

Analogous to our analysis of the other two audit studies, we first estimate a probit model of callback rates on implied race of the applicants and report the results in column 1 of Table A17 in the Online Appendix. Consistent with Nunley et al. (2015), we find that the callback rates for job applicants with African American-sounding names are 2.8 percentage points lower compared to those with White-sounding names. When applying our name fluency algorithm to the fictitious first and last names employed in Nunley et al. (2015), we observe in column 2 that the standardized algorithmic name rating is negatively and significantly correlated with callback rates.

In column 3, we include both race and name difficulty measures and find that the magnitude of the coefficient on being a Black applicant is reduced to -0.024 (p-value < 0.01), representing an approximately 15 percent decrease in the racial penalty estimated in column 1. Similar to our findings discussed in Section III.B, this implies that racial discrimination based on one's name partly works through the difficulty of pronouncing (and potentially processing and remembering) that name.

When controlling for name length, gender, as well as additional resume characteristics used in Nunley et al. (2015),[4] we show in column 4 that name difficulty is an important and significant factor in explaining the callback rates, with the coefficient now marginally significant at the 10 percent level, and the indicator variable for Black names remains negative and significant.

It is worth noting that the data from Nunley et al. (2015) uses only eight unique names (two for each gender-race combination), which are far fewer than the number used by Bertrand and Mullainathan (36) or Oreopoulos (44). Despite this important drawback and the resulting limited statistical power, our analysis of this additional experimental data confirms that name complexity is negatively related to the probability of receiving a callback and that an important channel for explaining name-based racial discrimination is through the fluency of one's name. These results are thus consistent with our main findings discussed in Section III.

Appendix E. Instructions for Name Fluency Surveys

Thank you for agreeing to assist with research projects related to the pronunciation of names. I have designed a set of Qualtrics surveys which have a series of names for you to pronounce.

1. Before you start a particular survey, start an audio recording of yourself. Then, you will see a series of names for you to pronounce, with one name per screen. Read through

[4] The set of resume characteristics includes college attended, academic major, grade point average, honor's distinction, employment status, socioeconomic status of the applicant's address, and dummies for month and city.

the name, and then click the arrow to advance to the next screen to see the next name. Continue to repeat this until you have finished the survey. You may then stop the recording and save it. You will repeat this process for all of the different groups of names, though you may wish to do break up your work across several different times in the day or the week to complete the work.

2. Please complete a particular group in one sitting without taking any breaks in between. Once you complete that group, then feel free to take as long of a break as you need until you start the next survey, but again, please do not take breaks once you have started a new survey until you complete that one. Names will be separated in groups of approximately 50 (with some groups listed as first names and some groups listed as last names), so perhaps you may want to do a bunch at one time, with short breaks in between each of the individual surveys. Then, you can come back and do another chunk of them at another day/time when you are free.

3. If you are unsure of how to pronounce a particular name, simply do your best to make a guess or sound it out before you click the arrow to advance to the next screen. You should not search the internet to hear a recording of the name, but simply make an attempt at pronouncing it.

4. It is possible that you may see some names that are duplicates or are very similar to other names in one of the surveys, but please pronounce each of the names you see on the screen even if you think you have seen that name before.

5. Please complete each survey only one time. To make sure that you do every survey only once, take careful notes about which ones you have completed and which ones you still need to complete. The most logical way would be to complete the surveys in numerical order (perhaps starting with the first names and then the last names).

References

Bertrand, M., and Mullainathan, S. (2004). "Are Emily and Greg more employable than Lakisha and Jamal? A field experiment on labor market discrimination." *American Economic Review*, *94*(4), 991–1013.

Ge, Q., Wu, S., and Zhou, C. (2021). "Sharing common roots: Student-graduate committee matching and job market outcomes." *Southern Economic Journal*, *88*(2), 828–856.

Nunley, J. M., Pugh, A., Romero, N., and Seals, R. A. (2015). "Racial discrimination in the labor market for recent college graduates: Evidence from a field experiment." *B.E. Journal of Economic Analysis & Policy*, *15*(3), 1093–1125.

Oreopoulos, P. (2011). "Why do skilled immigrants struggle in the labor market? A field experiment with thirteen thousand resumes." *American Economic Journal: Economic Policy*, *3*(4), 148–71.

Table A1: Name Fluency and Placement Outcomes: Alternative Algorithm Rating

	(1) Academia	(2) Academia	(3) TT	(4) TT	(5) RePEc_Imputed	(6) RePEc_Imputed
Alternative Algorithm Rating: Full Name	-0.040		-0.019		82.771	
	(0.016)		(0.017)		(31.479)	
Alternative Algorithm Rating: First Name		-0.037		-0.018		56.701
		(0.017)		(0.018)		(32.596)
Alternative Algorithm Rating: Last Name		-0.020		-0.009		68.384
		(0.019)		(0.019)		(36.238)
Observations	1,469	1,469	1,499	1,499	1,510	1,510
Control for Name Length	Yes	Yes	Yes	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes	Yes	Yes	Yes

Notes: The coefficients in columns 1-4 are marginal effects of probit regressions. The dependent variable in columns 1-2 (3-4) is a dichotomous variable for being placed in an academic (tenure track) position. Columns 5-6 are estimated using a tobit model, with the dependent variable being the imputed RePEc ranking of the institution of initial job placement, where private sector jobs are given an imputed ranking of $1,000$, the highest (worst) ranking. All tobit regressions are censored with an upper limit of $1,000$. The alternative algorithm rating for name pronunciation difficulty is based on an arithmetic average of the letter-based and phoneme-based sub-rating schemes. Robust standard errors are in parentheses.

Table A2: Name Fluency and Placement Quality: Tobit Estimates – Raw RePEc Ranking

	(1) RePEc	(2) RePEc	(3) RePEc
Algorithm Rating: Full Name	79.298		
	(24.802)		
Pronunciation Time: Full Name		51.069	
		(26.546)	
Subjectively Difficult: Full Name			28.278
			(49.645)
Observations	910	910	910
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes

Notes: The dependent variable across all specifications is the RePEc ranking of the institution of initial job placement, where individuals obtaining private sector jobs are excluded from the sample. All specifications are estimated using a tobit model censored with an upper limit of $1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. The pronunciation time rating is a survey-based measure that records the median time it takes individuals to pronounce a name. The subjective difficulty rating is based on individuals' independent subjective assessments. Robust standard errors are in parentheses.

Table A3: Name Fluency and Placement Types: Multinomial Logit Estimates

	(1) Academia	(2) Industry
Algorithm Rating: Full Name	-0.217	-0.161
	(0.101)	(0.120)
Observations	1,510	1,510
Control for Name Length	Yes	Yes
Other Controls	Yes	Yes
Subfield/Program FE	Yes	Yes
Region/JM Cycle FE	Yes	Yes

	(3) Academia	(4) Industry
Pronunciation Time: Full Name	-0.273	0.113
	(0.102)	(0.120)
Observations	1,510	1,510
Control for Name Length	Yes	Yes
Other Controls	Yes	Yes
Subfield/Program FE	Yes	Yes
Region/JM Cycle FE	Yes	Yes

	(5) Academia	(6) Industry
Subjectively Difficult: Full Name	-0.411	0.195
	(0.190)	(0.229)
Observations	1,510	1,510
Control for Name Length	Yes	Yes
Other Controls	Yes	Yes
Subfield/Program FE	Yes	Yes
Region/JM Cycle FE	Yes	Yes

Notes: Each panel is estimated using a separate multinomial logit model with the dependent variable being a categorical variable capturing placement types, including academia, government/think tank, and industry (private sector). Government/think tank positions are the baseline category across all specifications. The reported coefficients are in log-odds. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. The pronunciation time rating is a survey-based measure that records the median time it takes individuals to pronounce a name. The subjective difficulty rating is based on individuals' independent subjective assessments. Standard errors are in parentheses.

Table A4: Name Fluency and Placement Types: Multinomial Logit Estimates – Alternative Placement Categories

	(1) TT	(2) Govt/Think Tank	(3) Industry
Algorithm Rating: Full Name	0.100	0.284	0.119
	(0.099)	(0.125)	(0.117)
Observations	1,510	1,510	1,510
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes

	(4) TT	(5) Govt/Think Tank	(6) Industry
Pronunciation Time: Full Name	0.062	0.312	0.429
	(0.105)	(0.128)	(0.124)
Observations	1,510	1,510	1,510
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes

	(7) TT	(8) Govt/Think Tank	(9) Industry
Subjectively Difficult: Full Name	0.301	0.640	0.838
	(0.203)	(0.247)	(0.236)
Observations	1,510	1,510	1,510
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes

Notes: Each panel is estimated using a separate multinomial logit model with the dependent variable being a categorical variable capturing placement types, including tenure track, visiting/postdoc, government/think tank, and industry (private sector). Visiting/postdoc positions are the baseline category across all specifications. The reported coefficients are in log-odds. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. The pronunciation time rating is a survey-based measure that records the median time it takes individuals to pronounce a name. The subjective difficulty rating is based on individuals' independent subjective assessments. Standard errors are in parentheses.

Table A5: Name Fluency and Placement Quality: Ordered Probit Estimates

	(1) RePEc_Imputed	(2) RePEc_Imputed	(3) RePEc_Imputed
Algorithm Rating: Full Name	0.076		
	(0.046)		
Pronunciation Time: Full Name		0.100	
		(0.048)	
Subjectively Difficult: Full Name			0.106
			(0.087)
Observations	1,510	1,510	1,510
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes

Notes: All specifications are estimated using an ordered probit model, where the dependent variable is based on the following ordered categories of the imputed RePEc research productivity index: 1) RePEc ≤ 50; 2) $50 <$ RePEc ≤ 200; 3) $200 <$ RePEc ≤ 400; 4) $400 <$ RePEc ≤ 800; and 5) RePEc $= 1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. The pronunciation time rating is a survey-based measure that records the median time it takes individuals to pronounce a name. The subjective difficulty rating is based on individuals' independent subjective assessments. Robust standard errors are in parentheses.

Table A6: Name Fluency and Placement Outcomes: Controlling for Advisor Match

	(1) Academia	(2) Academia	(3) TT	(4) TT	(5) RePEc_Imputed	(6) RePEc_Imputed
Algorithm Rating: Full Name	-0.031	-0.030	-0.005	-0.005	66.572	65.118
	(0.016)	(0.016)	(0.017)	(0.017)	(31.722)	(31.664)
	(7) Academia	(8) Academia	(9) TT	(10) TT	(11) RePEc_Imputed	(12) RePEc_Imputed
Pronunciation Time: Full Name	-0.074	-0.074	-0.047	-0.046	80.610	80.334
	(0.018)	(0.018)	(0.018)	(0.018)	(33.735)	(33.952)
	(13) Academia	(14) Academia	(15) TT	(16) TT	(17) RePEc_Imputed	(18) RePEc_Imputed
Subjectively Difficult: Full Name	-0.117	-0.116	-0.060	-0.057	82.380	83.393
	(0.033)	(0.033)	(0.033)	(0.033)	(61.258)	(60.958)
Observations	1,469	1,469	1,499	1,499	1,510	1,510
Control for Country Match	Yes	No	Yes	No	Yes	No
Control for Language Match	No	Yes	No	Yes	No	Yes
Control for Name Length	Yes	Yes	Yes	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes	Yes	Yes	Yes

Notes: The coefficients in columns 1-4, 7-10, and 13-16 are marginal effects of probit regressions. The dependent variable in columns 1-2, 7-8, and 13-14 (3-4, 9-10 and 15-16) is a dichotomous variable for being placed in an academic (tenure track) position. Columns 5-6, 11-12, and 17-18 are estimated using a tobit model, with the dependent variable being the imputed RePEc ranking of the institution of initial job placement, where private sector jobs are given an imputed ranking of 1,000, the highest (worst) ranking. All tobit regressions are censored with an upper limit of $1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. The pronunciation time rating is a survey-based measure that records the median time it takes individuals to pronounce a name. The subjective difficulty rating is based on individuals' independent subjective assessments. The country/language match variables are indicator variables based on matching with at least one of the committee members. Robust standard errors are in parentheses.

Table A7: Name Fluency and Placement Outcomes by Gender

	Male Candidates			Female Candidates		
	(1) Academia	(2) TT	(3) RePEc_Imputed	(4) Academia	(5) TT	(6) RePEc_Imputed
Algorithm Rating: Full Name	-0.045	0.009	64.460	-0.036	-0.065	21.069
	(0.022)	(0.022)	(37.267)	(0.033)	(0.034)	(64.742)
Observations	970	1,016	1,053	392	413	457
Control for Name Length	Yes	Yes	Yes	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes	Yes	Yes	Yes

Notes: The coefficients in columns 1-2 and 3-4 are marginal effects of probit regressions. The dependent variable in columns 1 and 4 (2 and 5) is a dichotomous variable for being placed in an academic (tenure track) position. Columns 3 and 6 are estimated using a tobit model, with the dependent variable being the imputed RePEc ranking of the institution of initial job placement, where private sector jobs are given an imputed ranking of 1,000, the highest (worst) ranking. All tobit regressions are censored with an upper limit of $1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Robust standard errors are in parentheses.

Table A8: Name Fluency and Placement Outcomes: Excluding Candidates With Ethnically Chinese Names

	(1) Academia	(2) TT	(3) RePEc_Imputed
Algorithm Rating: Full Name	-0.037	-0.018	69.262
	(0.019)	(0.020)	(37.206)
Observations	1,094	1,093	1,131
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes

Notes: The sample excludes all job market candidates with ethnically Chinese names, regardless of their undergraduate locations. The coefficients in columns 1 and 2 are marginal effects of probit regressions. The dependent variable in column 1 (2) is a dichotomous variable for being placed in an academic (tenure track) position. Column 3 is estimated using a tobit model, with the dependent variable being the imputed RePEc ranking of the institution of initial job placement, where private sector jobs are given an imputed ranking of 1,000, the highest (worst) ranking. All tobit regressions are censored with an upper limit of $1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Robust standard errors are in parentheses.

Table A9: Name Fluency and Placement Outcomes: Country Fixed Effects

	(1) Academia	(2) TT	(3) RePEc_Imputed
Algorithm Rating: Full Name	-0.033	-0.002	76.923
	(0.017)	(0.018)	(31.260)
Observations	1,416	1,463	1,510
Control for Name Length	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes
Country/JM Cycle FE	Yes	Yes	Yes

Notes: The coefficients in columns 1 and 2 are marginal effects of probit regressions. The dependent variable in column 1 (2) is a dichotomous variable for being placed in an academic (tenure track) position. Column 3 is estimated using a tobit model, with the dependent variable being the imputed RePEc ranking of the institution of initial job placement, where private sector jobs are given an imputed ranking of 1,000, the highest (worst) ranking. All tobit regressions are censored with an upper limit of $1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Robust standard errors are in parentheses.

Table A10: Name Fluency and Placement Outcomes: Accounting for Common Names

	All Candidates			Candidates from U.S. and Canada		
	(1) Academia	(2) TT	(3) RePEc_Imputed	(4) Academia	(5) TT	(6) RePEc_Imputed
Common First Name	-0.006	-0.041	-44.623	-0.020	-0.040	20.921
	(0.043)	(0.045)	(89.256)	(0.058)	(0.054)	(115.622)
Common Last Name	0.006	-0.076	65.057	0.041	-0.044	101.178
	(0.069)	(0.069)	(124.601)	(0.106)	(0.098)	(172.005)
Common First Name ×	-0.232	-0.179	374.023	-0.119	-0.161	313.511
Common Last Name	(0.168)	(0.138)	(307.825)	(0.211)	(0.156)	(351.275)
Algorithm Rating: Full Name	-0.033	-0.014	69.557	-0.071	-0.048	130.065
	(0.017)	(0.017)	(32.978)	(0.029)	(0.028)	(55.243)
Observations	1,469	1,499	1,510	586	600	648
Control for Name Length	Yes	Yes	Yes	Yes	Yes	Yes
Other Controls	Yes	Yes	Yes	Yes	Yes	Yes
Subfield/Program FE	Yes	Yes	Yes	Yes	Yes	Yes
Region/JM Cycle FE	Yes	Yes	Yes	Yes	Yes	Yes

Notes: The coefficients in columns 1-2 and 4-5 are marginal effects of probit regressions. The dependent variable in columns 1 and 3 (2 and 5) is a dichotomous variable for being placed in an academic (tenure track) position. Columns 3 and 6 are estimated using a tobit model, with the dependent variable being the imputed RePEc ranking of the institution of initial job placement, where private sector jobs are given an imputed ranking of 1,000, the highest (worst) ranking. All tobit regressions are censored with an upper limit of $1,000$. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Common first and last names are derived from the 1990 and 2010 U.S. Census, respectively. Robust standard errors are in parentheses.

Table A11: Black/Ethnic Immigrant Names and Callback Rates in Bertrand and Mullainathan (2004) and Oreopoulos (2011)

Bertrand and Mullainathan (2004)

	Name Difficulty	Percent Callback
Black		
Ebony	-0.973	9.62
Kenya	-0.973	8.67
Leroy	-0.523	9.38
Tyrone	-0.361	5.33
Jermaine	0.004	9.62
Jamal	0.153	6.56
Tremayne	0.200	4.35
Tamika	0.297	5.47
Darnell	0.675	4.76
Rasheed	0.770	2.99
Latonya	0.826	9.13
Hakim	0.970	5.45
Kareem	1.038	4.69
Aisha	1.148	2.22
Keisha	1.547	3.83
Latoya	1.549	8.41
Tanisha	1.839	5.80
Lakisha	2.161	5.50
Average	0.575	6.21
Correlation:	-0.488 [0.040]	

Oreopoulos (2011)

	Name Difficulty	Percent Callback
Indian		
Tara Singh	-0.603	10.29
Maya Kumar	-0.538	8.66
Shreya Sharma	0.348	9.54
Arjun Kumar	0.742	7.82
Samir Sharma	0.985	8.59
Panav Singh	1.264	8.25
Rahul Kaur	1.913	8.14
Priyanka Kaur	2.557	7.61
Average:	0.834	8.61
Correlation:	-0.755 [0.030]	
Pakistani		
Hassan Khan	-0.304	6.30
Fatima Sheikh	0.245	8.11
Sana Khan	0.392	8.82
Ali Saeed	0.705	8.33
Chaudhry Mohammad	1.102	6.12
Asif Sheikh	1.296	3.85
Hina Chaudhry	1.348	7.80
Rabab Saeed	3.142	4.26
Average:	0.991	6.70
Correlation:	-0.588 [0.125]	
Chinese		
Na Li	-0.802	7.65
Min Liu	-0.671	11.34
Lei Li	-0.644	9.32
Tao Wang	-0.557	10.98
Dong Liu	-0.534	7.88
Fang Wang	-0.283	12.57
Yong Zhang	-0.279	8.60
Xiuying Zhang	1.511	7.42
Average:	-0.283	9.47
Correlation:	-0.338 [0.412]	
Indian/Pakistani/Chinese		
Average:	0.514	8.26
Correlation:	-0.594 [0.002]	

Notes: The table contains all Black and ethnic immigrant names taken from publicly available replication data for Bertrand and Mullainathan (2004) and Oreopoulos (2011), respectively. The reported correlations are between name difficulty ratings and callback rates. P-values for correlations are in brackets.

Table A12: Name Fluency and Callback Rates: Experimental Data from Bertrand and Mullainathan (2004)

	All Applicants				Black Applicants	
	(1) Callback	(2) Callback	(3) Callback	(4) Callback	(5) Callback	(6) Callback
Black	-0.032		-0.018	-0.015		
	(0.006)		(0.006)	(0.006)		
Female				0.005		0.012
				(0.006)		(0.006)
College Educated				0.007		0.012
				(0.008)		(0.007)
Number of Jobs on Resume				-0.002		0.002
				(0.003)		(0.003)
Years of Experience				0.008		0.003
				(0.001)		(0.001)
Years of Experience^2				-0.000		-0.000
				(0.000)		(0.000)
Honors				0.054		0.040
				(0.017)		(0.011)
Volunteering Experience				-0.002		0.008
				(0.008)		(0.010)
Military Experience				0.003		-0.013
				(0.015)		(0.008)
Working in School				-0.001		-0.006
				(0.003)		(0.004)
Listing Email				0.011		-0.003
				(0.008)		(0.008)
Computer Skills				-0.024		-0.009
				(0.011)		(0.009)
Special Skills				0.063		0.049
				(0.008)		(0.006)
First Name Length				0.003		0.006
				(0.003)		(0.003)
Algorithm Rating: First Name		-0.017	-0.011	-0.012	-0.011	-0.014
		(0.004)	(0.005)	(0.005)	(0.003)	(0.003)
Observations	4,870	4,870	4,870	4,870	2,435	2,435

Notes: The sample is derived from publicly available replication data for Bertrand and Mullainathan (2004). Columns 1-4 include all job applicants, while columns 5-6 focus on Black applicants. The reported coefficients are marginal effects of probit regressions, where the dependent variable is a dichotomous variable for receiving a callback. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Clustered standard errors at the job advertisement level are in parentheses.

Table A13: Name Fluency and Callback Rates: Experimental Data from Oreopoulos (2011)

	All Applicants					Ind/Pak/Chn Applicants	
	(1) Callback	(2) Callback	(3) Callback	(4) Callback	(5) Callback	(6) Callback	(7) Callback
Female				0.018	0.019		0.006
				(0.005)	(0.005)		(0.007)
Top 200 World Ranking University				-0.003	-0.003		0.006
				(0.005)	(0.005)		(0.007)
Listing Extracurricular Activities				-0.002	-0.002		0.011
				(0.005)	(0.005)		(0.006)
Fluent in French & Other Languages				0.019	0.019		0.021
				(0.007)	(0.007)		(0.009)
Master's Degree				0.006	0.006		0.007
				(0.007)	(0.007)		(0.010)
High Quality Work Experience				0.009	0.009		0.014
				(0.005)	(0.005)		(0.007)
Additional Required Credentials				0.041	0.041		0.024
				(0.014)	(0.014)		(0.015)
Listing Canadian References				-0.029	-0.028		-0.022
				(0.015)	(0.015)		(0.015)
Accreditation of Foreign Education				-0.012	-0.012		-0.006
				(0.013)	(0.013)		(0.013)
Permanent Resident				-0.007	-0.007		-0.007
				(0.014)	(0.014)		(0.013)
Indian	-0.046		-0.036	-0.035	-0.033		0.002
	(0.005)		(0.007)	(0.009)	(0.009)		(0.008)
Pakistani	-0.057		-0.049	-0.049	-0.050		-0.015
	(0.007)		(0.008)	(0.009)	(0.009)		(0.012)
Chinese	-0.041		-0.038	-0.035	-0.029		
	(0.005)		(0.006)	(0.009)	(0.011)		
Chinese Canadian	-0.053		-0.053	-0.050	-0.045		
	(0.006)		(0.006)	(0.007)	(0.008)		
Greek	-0.031		-0.018	-0.018	-0.035		
	(0.012)		(0.015)	(0.016)	(0.018)		
British	-0.024		-0.024	-0.023	-0.023		
	(0.008)		(0.008)	(0.008)	(0.008)		
Full Name Length				0.000			-0.000
				(0.002)			(0.002)
Algorithm Rating: Full Name		-0.014	-0.008	-0.007		-0.008	-0.007
		(0.003)	(0.004)	(0.004)		(0.003)	(0.004)
First Name Length					-0.001		
					(0.002)		
Last Name Length					0.003		
					(0.003)		
Algorithm Rating: First Name					-0.006		
					(0.004)		
Algorithm Rating: Last Name					-0.001		
					(0.004)		
Observations	12,910	12,910	12,910	12,910	12,910	7,158	7,158

Notes: The sample is derived from publicly available replication data for Oreopoulos (2011). Columns 1-5 include all job applicants, while columns 6-7 focus on applicants with ethnically Indian, Pakistani, and Chinese names. The reported coefficients are marginal effects of probit regressions, where the dependent variable is a dichotomous variable for receiving a callback. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Clustered standard errors at the job advertisement level are in parentheses.

Table A14: Name Fluency and Callback Rates: Experimental Data from Oreopoulos (2011) – Sample of Ethnic Immigrant Applicants

	Indian	Pakistani	Chinese
	(1)	(2)	(3)
	Callback	Callback	Callback
Algorithm Rating: Full Name	-0.006	-0.012	-0.031
	(0.006)	(0.009)	(0.019)
Observations	3,312	957	2,848
Control for Name Length	Yes	Yes	Yes
Control for Gender	Yes	Yes	Yes
Control for Resume Characteristics	Yes	Yes	Yes

Notes: The sample is derived from publicly available replication data for Oreopoulos (2011). All specifications in this table focus on job applicants with ethnically Indian, Pakistani, and Chinese names. The reported coefficients are marginal effects of probit regressions, where the dependent variable is a dichotomous variable for receiving a callback. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Clustered standard errors at the job advertisement level are in parentheses.

Table A15: Name Fluency and Callback Rates by Gender: Experimental Data from Bertrand and Mullainathan (2004) and Oreopoulos (2011)

Bertrand and Mullainathan (2004)	All Applicants		Black Applicants	
	Male	Female	Male	Female
	(1)	(2)	(3)	(4)
	Callback	Callback	Callback	Callback
Algorithm Rating: First Name	-0.006	-0.016	-0.019	-0.020
	(0.033)	(0.002)	(0.016)	(0.004)
Observations	1,124	3,746	549	1,886
Control for Name Length	Yes	Yes	Yes	Yes
Control for Race	Yes	Yes	No	No
Control for Resume Characteristics	Yes	Yes	Yes	Yes

Oreopoulos (2011)	All Applicants		Ind/Pak/Chn	
	Male	Female	Male	Female
	(5)	(6)	(7)	(8)
	Callback	Callback	Callback	Callback
Algorithm Rating: Full Name	-0.002	-0.014	-0.003	-0.013
	(0.011)	(0.006)	(0.011)	(0.006)
Observations	6,343	6,567	3,543	3,615
Control for Name Length	Yes	Yes	Yes	Yes
Control for Ethnicity	Yes	Yes	Yes	Yes
Control for Resume Characteristics	Yes	Yes	Yes	Yes

Notes: The samples are derived from publicly available replication data for Bertrand and Mullainathan (2004) and Oreopoulos (2011). Columns 1-2 and 5-6 are based on the full sample of each data set, while columns 3-4 and 7-8 focus on the sample of Black job applicants and applicants with ethnically Indian, Pakistani, and Chinese names, respectively. The reported coefficients are marginal effects of probit regressions, where the dependent variable is a dichotomous variable for receiving a callback. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Clustered standard errors at the job advertisement level are in parentheses.

Table A16: Name Fluency and Callback Rates: Experimental Data from Bertrand and Mullainathan (2004) and Oreopoulos (2011) – Sample of Low Quality Resumes

	Bertrand and Mullainathan (2004) Low Quality Resume			Oreopoulos (2011) No Master's			Pooled Data Low Quality Resume/No Master's		
	(1) Callback	(2) Callback	(3) Callback	(4) Callback	(5) Callback	(6) Callback	(7) Callback	(8) Callback	(9) Callback
Black	-0.023	0.001	0.007						
	(0.007)	(0.005)	(0.005)						
Algorithm Rating: First Name		-0.018	-0.021						
		(0.003)	(0.003)						
Indian				-0.047	-0.036	-0.035			
				(0.006)	(0.008)	(0.010)			
Pakistani				-0.058	-0.050	-0.050			
				(0.007)	(0.009)	(0.009)			
Chinese				-0.041	-0.038	-0.034			
				(0.006)	(0.006)	(0.010)			
Chinese Canadian				-0.058	-0.058	-0.055			
				(0.006)	(0.006)	(0.007)			
Greek				-0.020	-0.005	-0.006			
				(0.015)	(0.019)	(0.019)			
British				-0.025	-0.025	-0.024			
				(0.009)	(0.009)	(0.009)			
Algorithm Rating: Full Name					-0.009	-0.008			
					(0.004)	(0.005)			
Black/Immigrant (Ind/Pak/Chn)							-0.029	-0.015	-0.017
							(0.005)	(0.006)	(0.007)
Algorithm Rating: First Name								-0.012	-0.010
								(0.003)	(0.004)
Observations	2,424	2,424	2,424	10,717	10,717	10,717	13,141	13,141	13,141
Control for Name Length	No	No	Yes	No	No	Yes	No	No	Yes
Control for Gender	No	No	Yes	No	No	Yes	No	No	Yes
Control for Resume Characteristics	No	No	Yes	No	No	Yes	No	No	Yes

Notes: The samples are derived from publicly available replication data for Bertrand and Mullainathan (2004) and Oreopoulos (2011). All specifications in this table focus on the subsample of job applicants with low quality resumes, where resume quality is determined based on a subjective measure in Bertrand and Mullainathan (2004) and whether one holds a Master's degree in Bertrand and Mullainathan (2004). The reported coefficients are marginal effects of probit regressions, where the dependent variable is a dichotomous variable for receiving a callback. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Clustered standard errors at the job advertisement level are in parentheses.

Table A17: Name Fluency and Callback Rates: Experimental Data from Nunley et al. (2015)

	(1) Callback	(2) Callback	(3) Callback	(4) Callback
Black	-0.028		-0.024	-0.034
	(0.007)		(0.007)	(0.009)
Algorithm Rating: Full Name		-0.009	-0.005	-0.009
		(0.003)	(0.004)	(0.005)
Observations	9,396	9,396	9,396	9,396
Control for Name Length	No	No	No	Yes
Control for Gender	No	No	No	Yes
Control for Resume Characteristics	No	No	No	Yes

Notes: The sample is derived from replication data for Nunley et al. (2015). The reported coefficients are marginal effects of probit regressions, where the dependent variable is a dichotomous variable for receiving a callback. The algorithm rating for name pronunciation difficulty is based on a weighted average of the letter-based and phoneme-based sub-rating schemes, where the weights are derived from neural network learning. Clustered standard errors at the job advertisement level are in parentheses.

Technical Appendix for "How Do You Say Your Name? Difficult-To-Pronounce Names and Labor Market Outcomes"

Qi Ge* Stephen Wu†

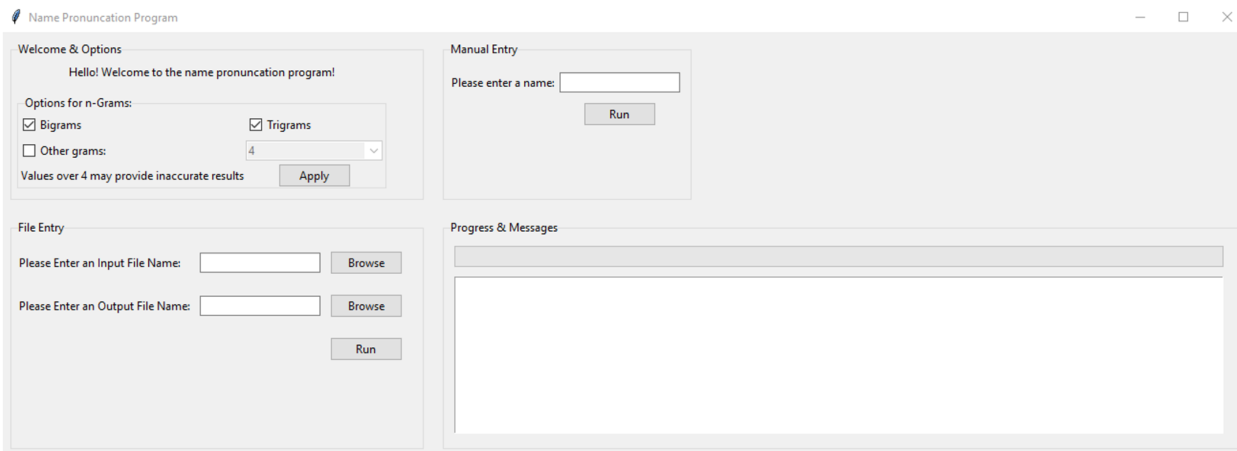
Abstract: In this technical appendix, we provide annotated code for the algorithm used to measure pronunciation difficulty for various words/names. This program was developed by James Kaffenbarger, Griffin Perry, Kenneth Talarico, Gwendolyn Urbanczyk, and Adam Valencia (December 2021).

*Department of Economics, Vassar College, Poughkeepsie, NY 12604, *qige@vassar.edu*.

†Department of Economics, Hamilton College, Clinton, NY 13323, *swu@hamilton.edu*.

Pronunciation Algorithm

To execute and load the interface that allows you to run the algorithm to measure word complexity, download the folder and then execute/open the file titled run.bat. The interface will look like:



Here is the python code for the main program:

```python
from nameui import *
from to_ipa import to_ipa
import csv
from NNModel import convertToModelFormat, get_parent_languge, get_combined_output
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import math
from random import choice
from ngrams import NgramManager, Ngrams
import os
import time


class MainModel:
    """ Class for the superclass that controls all of the main functionality and
        contains all of the other models as instance variables. """

```

```python
    def __init__(self, path_to_csv="ipa_dicts/english-general_american.csv"):
        """ Initializes models and the corpus of words. """
        with open(path_to_csv, encoding="utf8") as f:
            self.corpus = [w[1:-1] for row in csv.reader(f) \
                           for w in row[1].split(', ')]

        self.ipa_model = to_ipa(self)
        # SAE is "Standard American English"
        self.SAE_model = tf.keras.models.load_model('IsAmericanEnglishv4.0')
        self.root_model = tf.keras.models.load_model('RootLanguageModel')
        self.combine_model = tf.keras.models.load_model('Combine Scores Model')

        self.ngrams = NgramManager(self, 2, 3)

        # Needed to communicate/share data across threads
        self._gui = None
        self.prog_val = None
        self.to_gui_message = ""
        self.is_warning = False
        self.result = None
        self.lock = threading.Lock()

    def processInput(self, words):
        """ Method to be called every time the user submits new words. """

        # <names> is a list of every name the user inputted
        names = list(map(lambda x: x.lower().strip(), list(words[0])))
        self.addProgress(10)

        progressDivisor = len(names)
        if progressDivisor == 0:
            progressDivisor = len(names)

        # <ipa_names> is a list of the same length containing IPA transcriptions of each name
        #   i.e., ipa_names[i] is an IPA transcription of names[i]
        ipa_names = []
        progressVal = 0
        for name in names:
            ipa_names.append(self.ipa_model.to_ipa(name)[1:-1])

            progressVal += (15 / progressDivisor)
            if progressVal > 1:
```

```
                self.addProgress(int(progressVal))
                progressVal = 0

        self.sendToMessageLog("IPA conversion complete", False)

        gram_letters  = []
        progressVal = 0
        for name in names:
            gram_letters.append(round(100 - self.ngrams.generateLetterProbs(name), 2))

            progressVal += (10 / progressDivisor)
            if progressVal > 1:
                self.addProgress(int(progressVal))
                progressVal = 0

        gram_phonemes = []
        progressVal = 0
        for name in ipa_names:
            gram_phonemes.append(round(100 - self.ngrams.generatePhonemeProbs(name), 2))

            progressVal += (10 / progressDivisor)
            if progressVal > 1:
                self.addProgress(int(progressVal))
                progressVal = 0

        self.sendToMessageLog("N-gram calculations complete", False)

        # get neural net scores
        # Tnks seems to take a while?
        phonemeNN = convertToModelFormat(self.SAE_model,
                                         pd.read_csv('Eng_2Chars.csv'),
                                         self)
        rootLanguageNN = convertToModelFormat(self.root_model,
                                         pd.read_csv('singleChars.csv'),
                                         self)

        nn_scores = phonemeNN.convert(names)
        root_NN_scores = rootLanguageNN.convert(ipa_names)
        root_Parents = get_parent_languge(root_NN_scores)

        self.sendToMessageLog("Neural Network calculations complete", False)


```

```
        combinedNGrams = [round((gram_letters[i] + gram_phonemes[i]) / 2, 2)
                        for i in range(len(gram_letters))]

        final_scores = get_combined_output(self.combine_model, combinedNGrams, gram_letters, gram_phonemes, nn_scores)
        final_scores = [round(x, 2) for x in final_scores]
        self.sendToMessageLog("Final score calculations complete", False)

        # Threading Stuff - need to acquire the lock (just to make sure)
        # then write the dataframe to the result attribute before releasing
        # the lock and firing the end thread virtual event
        self.lock.acquire()
        self.result =  pd.concat([words[0],
                                  pd.DataFrame(final_scores),
                                  pd.DataFrame(gram_letters),
                                  pd.DataFrame(gram_phonemes),
                                  pd.DataFrame(nn_scores),
                                  pd.DataFrame(root_Parents)],
                                  axis=1, ignore_index=True)
        self.lock.release()
        self.addProgress(5)
        self._gui.generateEvent("<<ThreadEnded>>")

    def setGUI(self, gui_win):
        """
        Method used to set the object's gui attribute.
        @params - self
                - gui_win: the Root_Win object to set _gui to
        @returns - None
        """
        self._gui = gui_win


    def setNGrams(self, nlist):
        """
        Method used to set the object's NGram's manager object so the user
        can select which n they want to run with Ngrams. (This method cannot
        be run by the GUI while in a multithreaded state, that would
        probably create issues)
        @params - self
                - nlist: a list of ints to pass to the NGrams manager constructor
```

```python
        @returns - None
        """
        self.ngrams = NgramManager(self, *nlist)

    def addProgress(self, value):
        """
        Method used to add progress to the progress bar. Sets prog_val to value
        and then fires the virtual event to add progress
        @params - self
                - value: the value to add to the progress bar
        @returns - None
        """
        self.lock.acquire()
        self.prog_val = value
        self.lock.release()
        self._gui.generateEvent("<<AddProgress>>")

    def sendToMessageLog(self, output, warning=True):
        """
        Method used to output a message to the message log. Sets is_warning to
        warning, to_gui_message to output, and fires the
        <<SendMessage>> virtual event
        @params - self
                - output: The message to be outputted to the log
                - warning: If true, the message is treated as a warning.
                           Otherwise, it is treated as an 'info' message.
        @returns - None
        """
        self.lock.acquire()
        self.is_warning = warning
        self.to_gui_message = output
        self.lock.release()
        self._gui.generateEvent("<<SendMessage>>")


    def test_gui(self, words):
        """
        Method used to test the gui without running the entire program.
        To use, on the line root = RootWin(model), add a true parameter
        to the RootWin constructor.
        """
        self._gui.generateEvent("<<ThreadEnded>>")

```

```python
def main():
    """
    Main sets up the MainModel object and the GUI, then calls the GUI's
    mainloop.
    Since the GUI Needs to know about the model and the model about the GUI,
    we create the model first, then the GUI with the model, then set the model's
    gui to be the GUI we just created, before calling the mainloop.
    """
    try:
        model = MainModel()
        root = RootWin(model)
        model.setGUI(root)
    except Exception as e:
        output = "An error occured while setting up the program:\n"
        output += "".join(traceback.format_exception(type(e), e, e.__traceback__))
        print(output, file=sys.stderr)
        sys.exit(1)

    root.mainLoop()




if __name__ == '__main__':
    main()



# def testoutput():
#     with open("ipa_dicts/english-general_american.csv", encoding="utf8") as f:
#         reader = csv.reader(f)
#         corpus = [w[1:-1] for row in reader for w in row[1].split(', ')]
#     names = [choice(corpus) for _ in range(200)]
#     ipa_names = [ipa_model.ipa(name)[1:-1] for name in names]
#     ngrams_scores = [ngrams_phoneme_algorithm(name) for name in ipa_names]
#     nn_scores = getoutput(ipa_names, model)
#     final_scores = [round(((nn_scores[i] + ngrams_scores[i]) / 2) * 100, 2) for i in range(len(ngrams_scores))]
#     final_scores = [round(100 - x, 2) for x in final_scores]
#     with open("test-out.csv", 'w', encoding="utf8") as f:
```

```
#         writer = csv.writer(f)
#         writer.writerows([[names[i], final_scores[i]] for i in range(len(names))])
```

Here is python code that helps derive difficulty scores for letter n-grams and phoneme n-grams:

```
import csv
class NgramManager:
    def __init__(self, mainModel, *sizes):
        self.grams = [Ngrams(size) for size in sorted(sizes)]
        self.mainModel = mainModel

    def generateLetterProbs(self, words):
        probs = []
        #to deal with if name is multiple words
        words = words.split()
        for word in words:
            for gram in self.grams:
                if len(word) == 1:
                    #if the input is a single letter, "pronuncability" = 100
                    probs.append(100)
                if gram.length > len(word):
                    break
                probs.append(gram.generateLetterProbOccurence(word))
        if probs == []:
            self.mainModel.sendToMessageLog(f"Input: {word} too small for the current set nGrams, ignoring")
            return 0
        return sum(probs) / len(probs)

    def generatePhonemeProbs(self, words):
        probs = []
        #to deal with if name is multiple words
        words = words.split()
        for word in words:
            for gram in self.grams:
                if len(word) == 1:
                    #if the input is a single phoneme, "pronuncability" = 100
                    probs.append(100)
                if gram.length > len(word):
                    break
                probs.append(gram.generatePhonemeProbOccurence(word))
```

```
        if probs == []:
            self.mainModel.sendToMessageLog(f"Input: {word} too small for the current set nGrams, ignoring")
            return 0
        return sum(probs) / len(probs)

class Ngrams:
    def __init__(self, length, corpus="ipa_dicts/english-general_american.csv", occurence_table="unigram_freq.csv"):
        self.length = length
        self.corpus = corpus
        self.occurence_table = occurence_table
        self.letter_dictionary = {}
        self.phoneme_dictionary = {}
        self.letter_occurence_dictionary = {}
        self.phoneme_occurence_dictionary = {}
        self._generateNgramDictionaries()
        #self._generateOtherOccurrenceDictionaries()
        self._generateOccurrenceDictionaries()

    def _generateOtherOccurrenceDictionaries(self):
        """ Opens and creates dictionaries that map each gram in the occurence dictionary to
        how often it occurs, (most is 1, least is 0)"""
        #print("starting to generate dictionaries")
        with open(self.occurence_table, encoding="utf8") as f:
            for row in csv.reader(f):
                #row[0] is the word, row[1] is the phoneme, row[2] is the occurence value
                letter_grams = self.generateNgrams(row[0])
                phoneme_grams = self.generateNgrams(row[1])
                for gram in letter_grams:
                    if self.letter_occurence_dictionary.get(gram) is None:
                        self.letter_occurence_dictionary.update({gram: row[2]})
                    else:
                        num = self.letter_occurence_dictionary.get(gram)
                        self.letter_occurence_dictionary.update({gram: num + row[2]})
                #print("finished letter dictionaries")
                for gram in phoneme_grams:
                    if self.phoneme_occurence_dictionary.get(gram) is None:
                        self.phoneme_occurence_dictionary.update({gram: row[2]})
```

```
                    else:
                        num = self.phoneme_occurence_dictionary.get(gram)
                        self.phoneme_occurence_dictionary.update({gram: num + row[2]})

        #now we have the dictionaries with the total occurences. sort them from highest to lowest
        # and then scale them
        #print("generated non-scaled dictionaries")
        letter_sorted = sorted(self.letter_occurence_dictionary, key=self.letter_occurence_dictionary.get)
        for i in range(len(self.letter_occurence_dictionary)):
            self.letter_occurence_dictionary.update({letter_sorted[i]: ((i + 1) / len(self.letter_occurence_dictionary))})

        phoneme_sorted = sorted(self.phoneme_occurence_dictionary, key=self.phoneme_occurence_dictionary.get)
        for i in range(len(self.phoneme_occurence_dictionary)):
            self.phoneme_occurence_dictionary.update({phoneme_sorted[i]: ((i + 1) / len(self.phoneme_occurence_dictionary))})

        return

    def _generateOccurrenceDictionaries(self):
        """ Opens and creates dictionaries that map each word/phoneme to how often it occurs
             (most is 1, least is 0)"""
        count = 0
        with open(self.occurence_table, encoding="utf8") as f:
            for row in csv.reader(f):
                #hard coded the lengths of the occurence dictionaries, will need to change later
                #if user wants to provide their own
                self.letter_occurence_dictionary[row[0]] = ((333333 - count) / 333333)

                if self.phoneme_occurence_dictionary.get(row[1]) != None:
                    #this is done because there are a lot of words that are pronounced
                    #the same, but spelled differently
                    count += 1
                    continue
                self.phoneme_occurence_dictionary[row[1]] = ((333333 - count) / 333333)
                count += 1
```

```python

    def generateNgrams(self, str):
        """ Given a string and an n, return a list of all grams of that length"""
        answer = []
        for i in range(0, len(str) - self.length + 1):
            end = i + self.length
            answer.append(str[i:end])
        return answer

    def _generateNgramDictionaries(self):
        """ Generates the dictionaries for both letters and phonemes, keeping track of
            the total occurences"""
        with open(self.corpus, encoding="utf8") as f:
                letter_corpus = [w[1:-1] for row in csv.reader(f) \
                for w in row[0].split(', ')]
        with open(self.corpus, encoding="utf8") as f:
                phoneme_corpus = [w[1:-1] for row in csv.reader(f) \
                for w in row[1].split(', ')]

        for str in letter_corpus:
            letter_grams = self.generateNgrams(str)
            for gram in letter_grams:
                if self.letter_dictionary.get(gram) is None:
                    self.letter_dictionary.update({gram: 1})
                else:
                    num = self.letter_dictionary.get(gram)
                    self.letter_dictionary.update({gram: num + 1})

        for str in phoneme_corpus:
            phoneme_grams = self.generateNgrams(str)
            for gram in phoneme_grams:
                if self.phoneme_dictionary.get(gram) is None:
                    self.phoneme_dictionary.update({gram: 1})
                else:
                    num = self.phoneme_dictionary.get(gram)
                    self.phoneme_dictionary.update({gram: num + 1})

        return

    def generateDictionaryLetterProb(self, word):
        """ Given a word, scale data with 100 == most occurences in the dictionary,
            not to be confused with the occurence csv"""
```

```python
        grams = self.generateNgrams(word)
        max_occurences = max(self.letter_dictionary.values()) / 100
        average_gram_prob = 0
        for gram in grams:
            if self.letter_dictionary.get(gram) == None:
                #if the gram is not in the dictionary, treat it as zero to avoid
                #dividing NoneType
                continue
            average_gram_prob += self.letter_dictionary.get(gram) / max_occurences

        if average_gram_prob != 0:
            average_gram_prob = average_gram_prob / len(grams)
        return average_gram_prob

    def generateDictionaryPhonemeProb(self, word):
        """ Given a phoneme, scale data with 100 == most occurences in the dictionary,
            not to be confused with the occurence csv"""
        grams = self.generateNgrams(word)
        max_occurences = max(self.phoneme_dictionary.values()) / 100
        average_gram_prob = 0
        for gram in grams:
            if self.phoneme_dictionary.get(gram) == None:
                #if the gram is not in the dictionary, treat it as zero to avoid
                #dividing NoneType
                continue
            average_gram_prob += self.phoneme_dictionary.get(gram) / max_occurences

        if average_gram_prob != 0:
            average_gram_prob = average_gram_prob / len(grams)
        return average_gram_prob

    def generateLetterProbOccurence(self, word):
        """ Given a word, call generateDictionaryLetterProb, and then scale it up
            using the letter occurence table"""
        prob = self.generateDictionaryLetterProb(word)
        if self.letter_occurence_dictionary.get(word) == None:
            #word is not in the occurence dictionary, so no scaling is done
            return prob
```

```
        scaler = float(self.letter_occurence_dictionary[word])
        prob += (100 - prob) * scaler
        return prob

    def generatePhonemeProbOccurence(self, phoneme):
        """ Given a phoneme, call generateDictionaryPhonemeProb, and then scale it up
            using the phoneme occurence table"""
        prob = self.generateDictionaryPhonemeProb(phoneme)
        if self.phoneme_occurence_dictionary.get(phoneme) == None:
            #phoneme is not in the occurence dictionary, so no scaling is done
            return prob
        scaler = float(self.phoneme_occurence_dictionary[phoneme])
        prob += (100 - prob) * scaler
        return prob
```

The following code provides examples of calculation for a sample of words:

```
# def generateLetterProbOccurence(self, word):
#         """ Given a word, call generateDictionaryLetterProb, and then scale it up
#             using the letter occurence table"""
        # prob = self.generateDictionaryLetterProb(word)
        # average_scaler = 0
        # for gram in self.generateNgrams(word):
        #     if self.letter_occurence_dictionary.get(gram) == None:
        #         continue
        #     average_scaler += float(self.letter_occurence_dictionary[gram])
        # if average_scaler != 0:
        #     average_scaler = average_scaler / len(self.generateNgrams(word))
        # prob += (100 - prob) * average_scaler
        # return prob

# def generate_prob(self, word):
#     """ Given a word, compute the average gram prob """
#     grams = self.generateNgrams(word)
#     average_gram_prob = 0
#     for gram in grams:
#         average_gram_prob += self.dictionary.get(gram) / self.population

#     if average_gram_prob != 0:
```

```
#         average_gram_prob = average_gram_prob / len(grams)
#     return average_gram_prob

# data = ["hello", "world", "Ihope", "thisworks"]
# bi_gram = ngrams(data, 2)

# print(bi_gram.dictionary)
    # def ngrams_word_algorithm(word):
    #     """ Given a word, compute the tri_grams and get the average tri-gram value of the word
    #         from the corpus """
    #     word_trigrams = self.generateNgrams(word, 3)
    #     average_trigram_prob = 0
    #     for gram in word_trigrams:
    #         average_trigram_prob += tri_grams.get(gram) / bi_grams.get(gram[:-1])

    #     # To make sure that the word isn't composed completely of tri-grams not found
    #     # in the corpus
    #     if average_trigram_prob != 0:
    #         average_trigram_prob = average_trigram_prob / len(word_trigrams)

    #     return average_trigram_prob

    # def ngrams_phoneme_algorithm(phoneme):
    #     """ Given a phoneme, compute the z-score from the average of the bi-gram calculations
    #         and convert to a float between 0-1 """
    #     word_bigrams = generateNgrams(phoneme, 2)

    #     average_bigram_prob = 0
    #     for gram in word_bigrams:
    #         # If the corpus doesn't have this bi-gram, continue on to the next bi-gram.
    #         # Might need to change the weight of this later but for now it seems fine
    #         if bi_grams.get(gram) == None:
    #             continue

    #         average_bigram_prob += bi_grams.get(gram) / un_grams.get(gram[0])
    #         #average_bigram_prob += bi_grams.get(gram) / bi_gram_pop

```

```
    #     # To make sure that the word isn't composed completely of bi-grams not found
    #     # in the corpus
    #     if average_bigram_prob != 0:
    #         average_bigram_prob = average_bigram_prob / len(word_bigrams)

    #     z_score = (average_bigram_prob - average_corpus_prob) / standard_deviation

    #     answer = .5 * (math.erf(z_score / 2 ** .5) + 1) # https://stackoverflow.com/questions/2782284/function-to-convert-a-z-score-into-a-percentage

    #     return answer #average_bigram_prob
    #average_corpus_prob = len(bi_grams) / bi_gram_pop
# average_corpus_prob = 0
# for gram in bi_grams:
#     average_corpus_prob += bi_grams.get(gram) / un_grams.get(gram[0])
# average_corpus_prob = average_corpus_prob / bi_gram_pop

# standard_deviation = 0
# for gram in bi_grams:
#     standard_deviation += (bi_grams.get(gram) / un_grams.get(gram[0]) - average_corpus_prob) * (bi_grams.get(gram) / un_grams.get(gram[0]) - average_corpus_prob)
#     #standard_deviation += ((bi_grams.get(gram) / bi_gram_pop) - average_corpus_prob) * ((bi_grams.get(gram) / bi_gram_pop) - average_corpus_prob)
# standard_deviation = standard_deviation / (bi_gram_pop - 1)
# standard_deviation = math.sqrt(standard_deviation)
```

The following code takes the letter-based difficulty scores and the phoneme-based difficulty scores and uses a neural network model to calculate a final word difficulty score that is scaled to be between 0-100:

```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
import pandas as pd
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
import numpy as np
import time
```

```python
import os
from to_ipa import to_ipa

class convertToModelFormat():
    def __init__(self, model, columns, mainModel):
        self.model = model
        self.columns = columns
        self.columns.columns = ["Char(s)"]
        self.mainModel = mainModel



    def convert(self, inputlist):
        """Takes in inputs, and uses the columns given by preselected csv to run on the matching model
        """
        output = []
        progressDivisor = len(inputlist)
        if progressDivisor == 0:
            progressDivisor = len(inputlist)

        progressVal = 0
        temparr = []

        for ipaword in inputlist:

            temp = []
            for i in self.columns['Char(s)']:

                if i in ipaword:
                    temp.append(1)
                else:
                    temp.append(0)

            temparr.append(temp)
            progressVal += 25 / progressDivisor
            if progressVal > 1:
                self.mainModel.addProgress(int(progressVal))
                progressVal = 0


        answer = pd.DataFrame(temparr)
        answer.columns = self.columns['Char(s)'].values

        # Most of the runtime, presumably. Unpack?
```

```python
        prediction = self.model.predict(temparr)


        roundedpred = []
        for i in prediction:
            temp = []
            for j in i:
                temp.append(j.round())
            roundedpred.append(temp)

        #output.append(roundedpred)

        return roundedpred

def get_parent_languge(arr):
    outputs = []
    for i in arr:
        if i[0] == 1:
            outputs.append("Germanic")
        elif i[1] == 1:
            outputs.append("Romance")
        elif i[2] == 1:
            outputs.append("Sino-Tebetan")
        else:
            outputs.append("Japonic")
    return outputs

def get_combined_output(model, final_scores, gram_letters, gram_phonemes, nn_scores):
    """Takes in the model, and the outputs from all other aspects of the program, and combines them into one score"""
    #The STDDEV and mean of the training data, used for scaling the outputs
    STDDEV = 0.136461
    MEAN = 1.251892
    inputDF = pd.DataFrame()
    temp = []
    for i in nn_scores:
        temp.append(i[0])
    inputDF["FinScores"] = final_scores
    inputDF["LetterNGramScores"] = gram_letters
    inputDF["PhonemeNGramScores"] = gram_letters
    inputDF["NNscores"] = temp
    prediction = model.predict(inputDF)
    holder = []
```

```python
    for i in prediction:
        for j in i:
            #Ensures score is never over 100 or below 0
            if ((((j-MEAN)/STDDEV)*33) +50)> 100:
                holder+=[100.0]
            elif ((((j-MEAN)/STDDEV)*33) + 50)< 0:
                holder+=[0.0]
            else:
                #Scaled by 33 to make results spread wider across all
                ↪  values between 0-100, not centered around 50
                holder+=[(((j-MEAN)/STDDEV)*33) + 50]

    return holder
```